HZ BOOKS

软件工程技术丛书

测试系列

软件测试

（原书第2版）

Software Testing
A Craftsman's Approach
(Second Edition)

（美） Paul C. Jorgensen 著　　韩柯 杜旭涛 译

56

CRC PRESS

机械工业出版社
China Machine Press

VISIT…

LANZAROTE Caliente.com

Software Testing
A Craftsman's Approach *(Second Edition)*

软件测试 （原书第2版）

本书是经典的软件测试教材，是ACM&IEEE编制“软件工程知识体系”(SWEBOK)的主要参考文献之一，并已被国际众多大学选作教材。书中全面地介绍了软件测试的基础知识和方法，很好地做到了理论与实践相结合。

主要特点：

- 使用了独立于具体编程语言的伪代码
- 将UML集成到面向对象测试中
- 提供了大量的图表和案例研究
- 专门讲述了GUI测试方面的内容

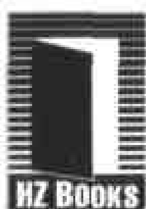
作者简介：

Paul C. Jorgensen 博士在其职业生涯的前20年中，主要从事电话交换系统的开发、支持和测试工作。1986年以来，他一直在大学为研究生讲授软件工程课程，先是在亚利桑那州立大学，然后在大峡谷州立大学。他的电子邮件地址是：jorgensp@gvsu.edu。

适用水平：中级

封面制作 锡彬





华章图书



网上购书：www.china-pub.com
北京市西城区百万庄南街1号 100037
读者服务热线：(010)68995259，68995264
读者服务信箱：hzedu@hzbook.com
http://www.hzbook.com
ISBN 7-111-12166-X/TP · 2676
定价：35.00 元

软件工程技术丛书

试系列

TP311.56
122

（美） Paul C. Jorgensen 著
韩柯 杜旭涛 译

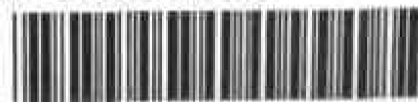
软件测试

（原书第2版）

Software Testing
A Craftsman's Approach
(Second Edition)



机械工业出版社



Z106686
北京信息工程学院图书馆

本书全面地介绍了软件测试的基础知识和方法，通过问题、图表和案例研究，对软件测试数学问题和技术进行了深入的研究，并在例子中以更加通用的伪代码取代了过时的Pascal代码，从而使内容独立于具体的程序设计语言。本书还介绍了面向对象测试的内容，并完善了GUI测试内容。

本书是ACM与IEEE计算机学会“软件工程知识体系”主要引用文献，并是国际众多大学的教材。

本书适合作为相关专业高校教材，也可用于读者自学。

Paul C. Jorgensen: Software Testing: A Craftsman's Approach (Second Edition) (ISBN: 0-8493-0809-7).

Original English language edition published by CRC Press LLC, 2000 NW Corporate Boulevard, Boca Raton, Florida 33431, USA.

Copyright © 2002 by CRC Press LLC.

Simplified Chinese language edition copyright © 2003 by China Machine Press.

All rights reserved.

本书中文版由美国CRC出版公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究

本书版权登记号：图字：01-2003-0268

图书在版编目（CIP）数据

软件测试（原书第2版）/（美）乔根森（Jorgensen, P. C.）著；韩柯等译. -北京：机械工业出版社，2003.7

（软件工程技术丛书 测试系列）

书名原文：Software Testing: A Craftsman's Approach (Second Edition)

ISBN 7-111-12166-X

Ⅰ.软… Ⅱ.①乔… ②韩… Ⅲ.软件－测试 Ⅳ.TP311.5

中国版本图书馆CIP数据核字（2003）第043580号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：杨 文

北京昌平奔腾印刷厂印刷·新华书店北京发行所发行

2003年7月第1版第1次印刷

787mm×1092mm 1/16 · 20.75印张

印数：0 001-5 000册

定价：35.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

译者序

软件测试是软件质量保证的重要内容，随着软件规模的不断扩大，复杂程度的不断提高，以及面向对象程序设计方法和工具的使用，软件测试的难度也进一步提高，测试质量更加难以度量。

本书的内容是以作者长期从事软件工程和软件测试研究生教学的讲义为基础，而作者的学生大部分具有一定的程序设计实际经验。作者的教学讲义经过长时间的教学实践，广泛采纳了学生的意见，既具有鲜明的教科书特点，又具有很强的实践性，既详细介绍了软件测试的基础理论和原理，也反映出作者对软件测试理论和实践的独到见解。本书的第1版在1995年出版，是ACM和IEEE计算机学会（www.swebok.org）联合编写的“软件工程知识体系”试用标准的主要引用文献之一，可见本书具有相当高的权威性。

在翻译过程中，除了对原文个别明显文字错误进行了相应更正外，我们力求忠实原文但由于译者的知识水平和实际工作经验有限，不当之处在所难免，恳请读者批评指正。参加本书翻译、审校和其他辅助工作的还有：吴铃、张红旭、原小铃、李津津、王威、屈健、黄惠菊、韩文臣、朱军、杜蔚轩、解冀海、付程、孟海军、耿民等。

译者

2003年3月

第1版前言

我们聚在会议室门口窃窃私语，纷纷通过门上的小窗户向里观望。会议室里，一位新来的软件设计师正在会议桌上摊开源程序清单，在水晶灯下小心翼翼地检查源代码，并不时地在清单上用红笔勾画。事后，我的一个同事问这位设计师在会议室做什么。他冷冷地说："找我程序中的bug。"这是发生在20世纪80年代中期的一件真人真事，那时人们很相信隐藏在水晶灯中的魔力。

本书的目标就是向读者提供一盏更好的水晶灯。我认为软件（和系统）测试是一种工艺，我觉得我已经在一定程度上掌握了这种工艺。在我多年开发电话交换机系统的经历中，大约有三分之一的时间花在了测试上：定义测试方法论和标准，协调主要国际长途收费电话交换机的系统测试、描述并帮助构建两个测试执行工具（现在我们称之为CASE工具），以及相当多的平淡无奇的手工测试。过去七年来，我在大学为研究生讲授软件工程，学术研究主要集中在规格说明和测试上。与Oxford Method（牛津方法）所说的一样，只有在能够讲授之后才算真正学会，我觉得一点不错。参加我的测试研究生课程的学生，都是当地公司的全职员工。说真的，这些学生会使人更加将理论联系于实际。本书就是根据我的讲义和工程经验总结出来的。

我自认为是一名软件工程师，但是在传统软件领域里我的知识精度和深度使我对这个头衔感到有点不自在。当Myers的著作《软件测试的艺术》（The Art of Software testing）刚出版时，我和我的一位同事正要返回意大利完成一个项目。在去机场的路上，我们走进一家麻省理工学院书店购买了此书，从而成为这本书最早的几个购买者之一。在过去的15年中，我们相信软件测试已经从一门艺术走向一种工艺，但是在成为一门科学之前，还有一段路要走

所有工艺的一部分都是了解工具和媒介的能力和局限性。好的木工有各种工具，根据要制做的东西和要使用的木材来选用最适合的工具。在软件开发生命周期传统瀑布模型的各个阶段，测试对于精确分析具有最重要的作用。将软件测试提升为一种工艺，要求测试工艺师了解基本工具。为此，本书第3章和第4章将提供在本书其他部分中使用的数学背景知识。

数学是一种描述工具，有助于人们更好地理解要被测试的软件。仅拥有精确表示法本身还不够，还必须拥有好的技术和判断方法以确定恰当的测试方法、并将其付诸实现。这些就是本书第二部分和第三部分的目标，这两个部分将讨论基本功能和结构性测试技术。这些技术将用于第2章介绍的系列例子。第四部分将这些技术用于集成和系统级测试，以及面向对象的测试。在这些层次上，更重要的是测试什么，而不是怎样测试，因此讨论的重点是测试需求规格说明。第四部分的最后，将研究软件控制系统中交互的测试，并简要讨论客户-服务器系统。

具有讽刺意味的是有关测试的书籍也会存在错误。尽管评审人员和编辑付出了艰辛的劳

动，但是我敢肯定书中还是会有错误，这些都要由我负责

1997年我参加了Edward Miller组织的测试研讨会，从那以后，Edward Miller成为软件测试界的领袖之一。在那次研讨会上，Miller详细地说明了测试工作不一定是十分繁琐的苦差，而可以是软件开发中富有创造性、很有意思的部分，本书的目标就是使读者成为测试工艺师，使读者能够通过出色的测试工作获得真正工艺师的自豪感和乐趣

Paul C. Jorgensen
密歇根州Rockford
1995年1月

前　　言

自从我为本书第1版撰写前言以来已经过去了七年　这七年中发生了很多事情，因此需要编写第2版．最重要的变化是UML（统一建模语言）已经成为描述和设计面向对象软件的一种标准。第2版的主要变化是增加了第五部分（共5章）专门讨论测试面向对象软件　第五部分中的绝大部分内容都以UML为基础

第二个主要变化是第1版中的所有Pascal例子替换为独立于语言的伪代码　大多数例子都做了详细描述，都可以由CRC出版公司的Web站点（www.crcpress.com）的Visual Basic可执行模块支持。新增加了一些说明测试面向对象软件的例子，对老例子做了几十处修改　最重要的补充是增加了经过改进的等价类测试描述、一个系列案例研究和集成测试的更多细节。

我很高兴本书第1版成为由ACM和IEEE计算机学会（www.swebok.org）联合编写的"软件工程知识体系"试用标准的主要引用文献之一，而这又使改正第1版中存在的问题变得更为重要。韩国的一位读者给我寄来第1版的错误清单，有38个错误、令人高兴的是，参加我讲授的软件测试研究生课程的学生也找出了其他一些错误　这和测试过程很相似；我改正了所有已知的错误。如果读者发现了错误，请通知我．我会对这些错误负责　我的电子邮件地址是：jorgensp@gvsu.edu

我要感谢CRC出版公司Jerry Papke和Helena Redshaw的认真而耐心的工作，还要感谢我的朋友和同事Roger Ferguson教授，感谢他对第五部分新内容提供的帮助，特别是在面向对象日历例子上提供的帮助。在某种意义上，Roger是第16~20章大量草稿的测试者

Paul C. Jorgensen
密歇根州Rockford
2002年5月

目　录

译者序

第1版前言

前言

第一部分　数学背景

第1章　测试概述 ……2

1.1　基本定义 ……2

1.2　测试用例 ……3

1.3　通过维恩图理解测试 ……4

1.4　标识测试用例 ……6

1.4.1　功能性测试 ……6

1.4.2　结构性测试 ……7

1.4.3　功能性测试与结构性测试的比较 ……8

1.5　错误与缺陷分类 ……9

1.6　测试级别 ……11

1.7　参考文献 ……12

1.8　练习 ……12

第2章　举例 ……13

2.1　泛化的伪代码 ……13

2.2　三角形问题 ……15

2.2.1　问题陈述 ……15

2.2.2　讨论 ……15

2.2.3　传统实现 ……15

2.2.4　结构化实现 ……18

2.3　NextDate函数 ……20

2.3.1　问题陈述 ……20

2.3.2　讨论 ……20

2.3.3　实现 ……20

2.4　佣金问题 ……23

2.4.1　问题陈述 ……23

2.4.2　讨论 ……23

2.4.3　实现 ……24

2.5　SATM系统 ……24

2.5.1　问题陈述 ……25

2.5.2　讨论 ……27

2.6　货币转换器 ……27

2.7　土星牌挡风玻璃雨刷 ……28

2.8　参考文献 ……28

2.9　练习 ……28

第3章　测试人员的离散数学 ……30

3.1　集合论 ……30

3.1.1　集合成员关系 ……30

3.1.2　集合定义 ……31

3.1.3　空集 ……31

3.1.4　维恩图 ……32

3.1.5　集合操作 ……32

3.1.6　集合关系 ……34

3.1.7　子集划分 ……34

3.1.8　集合恒等式 ……35

3.2　函数 ……36

3.2.1　定义域与值域 ……36

3.2.2　函数类型 ……37

3.2.3　函数合成 ……38

3.3　关系 ……39

3.3.1　集合之间的关系 ……39

3.3.2　单个集合上的关系 ……40

3.4　命题逻辑 ……41

3.4.1　逻辑操作符 ……42

3.4.2　逻辑表达式 ……42

3.4.3　逻辑等价 ……43

3.5　概率论 ……44

3.6　参考文献 ……45

3.7　练习 ……45

第4章　测试人员的图论 ……47

4.1　图 ……47

4.1.1　节点的度 ……48

4.1.2 关联矩阵 ……48
4.1.3 相邻矩阵 ……49
4.1.4 路径 ……49
4.1.5 连接性 ……50
4.1.6 压缩图 ……51
4.1.7 圈数 ……51
4.2 有向图 ……52
4.2.1 内度与外度 ……53
4.2.2 节点的类型 ……53
4.2.3 有向图的相邻矩阵 ……54
4.2.4 路径与半路径 ……54
4.2.5 可到达性矩阵 ……55
4.2.6 n-连接性 ……55
4.2.7 强组件 ……56
4.3 用于测试的图 ……57
4.3.1 程序图 ……57
4.3.2 有限状态机 ……58
4.3.3 Petri网 ……60
4.3.4 事件驱动的Petri网 ……62
4.3.5 状态图 ……65
4.4 参考文献 ……67
4.5 练习 ……67

第二部分 功能性测试

第5章 边界值测试 ……70
5.1 边界值分析 ……70
5.1.1 归纳边界值分析 ……71
5.1.2 边界值分析的局限性 ……72
5.2 健壮性测试 ……73
5.3 最坏情况测试 ……73
5.4 特殊值测试 ……74
5.5 举例 ……75
5.5.1 三角形问题的测试用例 ……75
5.5.2 NextDate函数的测试用例 ……79
5.5.3 佣金问题的测试用例 ……82
5.6 随机测试 ……84
5.7 边界值测试的指导方针 ……85
5.8 练习 ……86
第6章 等价类测试 ……87
6.1 等价类 ……87
6.1.1 弱一般等价类测试 ……88
6.1.2 强一般等价类测试 ……88
6.1.3 弱健壮等价类测试 ……89
6.1.4 强健壮等价类测试 ……90
6.2 三角形问题的等价类测试用例 ……90
6.3 NextDate函数的等价类测试用例 ……92
6.4 佣金问题的等价类测试用例 ……95
6.4.1 输出值域等价类测试用例 ……96
6.4.2 输出值域等价类测试用例 ……97
6.5 指导方针和观察 ……97
6.6 参考文献 ……98
6.7 练习 ……98
第7章 基于决策表的测试 ……100
7.1 决策表 ……100
7.2 三角形问题的测试用例 ……104
7.3 NextDate函数测试用例 ……105
7.3.1 第一次尝试 ……105
7.3.2 第二次尝试 ……106
7.3.3 第三次尝试 ……108
7.4 佣金问题的测试用例 ……110
7.5 指导方针与观察 ……110
7.6 参考文献 ……111
7.7 练习 ……111
第8章 功能性测试回顾 ……112
8.1 测试工作量 ……112
8.2 测试效率 ……115
8.3 测试的有效性 ……115
8.4 指导方针 ……116
8.5 案例研究 ……117

第三部分 结构性测试

第9章 路径测试 ……124
9.1 DD-路径 ……126
9.2 测试覆盖指标 ……129
9.2.1 基于指标的测试 ……129
9.2.2 测试覆盖分析器 ……131
9.3 基路径测试 ……131
9.3.1 McCabe的基路径方法 ……132

9.3.2 关于McCabe基路径方法的观察 …134
9.3.3 基本复杂度 ……136
9.4 指导方针与观察……138
9.5 参考文献 ……140
9.6 练习 ……141
第10章 数据流测试 ……143
10.1 定义/使用测试……143
10.1.1 举例 ……144
10.1.2 stocks的定义-使用路径 ……148
10.1.3 locks的定义-使用路径 ……148
10.1.4 totalLocks的定义-使用路径 ……149
10.1.5 sales的定义-使用路径 ……149
10.1.6 commission的定义-使用路径 ……150
10.1.7 定义-使用路径测试覆盖指标 ……151
10.2 基于程序片的测试 ……152
10.2.1 举例 ……154
10.2.2 风格与技术……157
10.3 指导方针与观察 ……158
10.4 参考文献 ……159
10.5 练习……159
第11章 结构性测试回顾 ……160
11.1 漏洞与冗余 ……160
11.2 用于方法评估的指标 ……162
11.3 重温案例研究 ……164
11.3.1 基于路径的测试 ……167
11.3.2 数据流测试……167
11.3.3 片测试 ……167
11.4 参考文献 ……167
11.5 练习……168

第四部分 集成与系统测试

第12章 测试层次 ……170
12.1 测试层次的传统观点 ……170
12.2 其他生命周期模型 ……171
12.2.1 瀑布模型的新模型 ……172
12.2.2 基于规格说明的生命周期模型 …173
12.3 ASTM系统 ……175
12.4 将集成测试与系统测试分开……184
12.4.1 结构认识 ……185
12.4.2 行为认识 ……186
12.5 参考文献 ……186
第13章 集成测试 ……187
13.1 深入研究SATM系统……187
13.2 基于分解的集成 ……191
13.2.1 自顶向下集成……191
13.2.2 自底向上集成……192
13.2.3 三明治集成……193
13.2.4 优缺点 ……194
13.3 基于调用图的集成 ……194
13.3.1 成对集成 ……194
13.3.2 相邻集成 ……194
13.3.3 优缺点 ……196
13.4 基于路径的集成 ……197
13.4.1 新概念与扩展概念 ……197
13.4.2 SATM系统中的MM-路径 ……200
13.4.3 MM-路径复杂度 ……203
13.4.4 优缺点 ……204
13.5 案例研究……205
13.5.1 基于分解的集成 ……209
13.5.2 基于调用图的集成 ……209
13.5.3 基于MM-路径的集成……209
13.6 参考文献……210
13.7 练习……210
第14章 系统测试 ……211
14.1 线索……211
14.1.1 线索的可能性……212
14.1.2 线索定义 ……213
14.2 需求规格说明的基本概念……214
14.2.1 数据 ……215
14.2.2 行动 ……215
14.2.3 设备 ……215
14.2.4 事件 ……216
14.2.5 线索 ……216
14.2.6 基本概念之间的关系 ……216
14.2.7 采用基本概念建模 ……217
14.3 寻找线索……219
14.4 线索测试的结构策略 ……222

14.4.1 自底向上组织线索 ……………………223
14.4.2 节点与边覆盖指标 ……………………224
14.5 线索测试的功能策略 ……………………225
14.5.1 基于事件的线索测试 ……………………225
14.5.2 基于端口的线索测试 ……………………227
14.5.3 基于数据的线索测试 ……………………227
14.6 SATM测试线索 ……………………229
14.7 系统测试指导方针 ……………………233
14.7.1 伪结构系统测试 ……………………233
14.7.2 运行剖面 ……………………233
14.7.3 累进测试与回归测试 ……………………235
14.8 参考文献 ……………………236
14.9 练习 ……………………236
第15章 交互测试 ……………………237
15.1 交互的语境 ……………………237
15.2 交互的分类 ……………………239
15.2.1 单处理器中的静态交互 ……………………240
15.2.2 多处理器中的静态交互 ……………………241
15.2.3 单处理器中的动态交互 ……………………242
15.2.4 多处理器中的动态交互 ……………………247
15.3 交互、合成与确定性 ……………………254
15.4 客户-服务器测试 ……………………256
15.5 参考文献 ……………………257
15.6 练习 ……………………257

第五部分 面向对象的测试

第16章 面向对象的测试问题 ……………………260
16.1 面向对象测试的单元 ……………………260
16.2 合成与封装的涵义 ……………………261
16.3 继承的涵义 ……………………263
16.4 多态性的涵义 ……………………264
16.5 面向对象测试的层次 ……………………264
16.6 GUI测试 ……………………264
16.7 面向对象软件的数据流测试 ……………………265
16.8 第五部分采用的例子 ……………………265
16.8.1 面向对象的日历 ……………………265
16.8.2 货币转换应用程序 ……………………266
16.9 参考文献 ……………………270
16.10 练习 ……………………270
第17章 类测试 ……………………271
17.1 以方法为单元 ……………………271
17.1.1 o-oCalendar的伪代码 ……………………272
17.1.2 Date.increment的单元测试 ……………………276
17.2 以类为单元 ……………………277
17.2.1 windshieldWiper类的伪代码 ……………………277
17.2.2 windshieldWiper类的单元测试 ……………………278
第18章 面向对象的集成测试 ……………………282
18.1 集成测试的UML支持 ……………………282
18.2 面向对象软件的MM-路径 ……………………284
18.3 面向对象数据流集成测试框架 ……………………290
18.3.1 事件驱动和消息驱动的Petri网 ……………………291
18.3.2 由继承导出的数据流 ……………………292
18.3.3 由消息导出的数据流 ……………………292
18.3.4 分片 ……………………294
18.4 练习 ……………………294
18.5 参考文献 ……………………296
第19章 GUI测试 ……………………297
19.1 货币转换程序 ……………………297
19.2 货币转换程序的单元测试 ……………………301
19.3 货币转换程序的集成测试 ……………………302
19.4 货币转换程序的系统测试 ……………………303
19.5 练习 ……………………307
第20章 面向对象的系统测试 ……………………308
20.1 货币转换器的UML描述 ……………………308
20.1.1 问题陈述 ……………………308
20.1.2 系统功能 ……………………308
20.1.3 表示层 ……………………309
20.1.4 高层用例 ……………………309
20.1.5 基本用例 ……………………310
20.1.6 详细GUI定义 ……………………311
20.1.7 扩展基本用例 ……………………312
20.1.8 真实用例 ……………………315
20.2 基于UML的系统测试 ……………………315
20.3 基于“状态图”的系统测试 ……………………318
20.4 参考文献 ……………………318

第一部分

数 学 背 景

第1章
测试概述

我们为什么要测试？两个主要原因是：对质量或可接受性做出判断，以及发现问题。我们进行测试，是因为知道我们很容易犯错误，特别是在软件领域和软件控制的系统中。本章的目标是对软件测试的背景进行概要介绍。以后的内容将围绕这个背景展开

1.1 基本定义

很多测试文献都陷入了混乱的（有时是不一致的）术语泥潭，也许因为测试技术在过去几十年中通过一些作者的总结已经得到发展。本书通篇使用的术语都采用电子电气工程师学会（IEEE）计算机协会所制定的标准。首先介绍几个有用的术语。

错误（error）——人类会犯错误。很接近的一个同义词是过错（mistake）。人们在编写代码时会出现过错，我们把这种过错叫做bug。错误很可能扩散，需求错误在设计期间有可能被放大，在编写代码时还会进一步扩大。

缺陷（fault）——缺陷是错误的结果。更精确地说，缺陷是错误的表现，而表现是表示的模式，例如叙述性文字、数据流框图、层次结构图、源代码等。与缺陷很接近的一个同义词是缺点（defect），程序错误也是。缺陷可能很难捕获。当设计人员出现遗漏错误时，所导致的缺陷会是遗漏本来应该在表现中提供的内容。这种情况说明需要对定义做进一步的细化，借用教堂常用的一个词，我们可以把缺陷分为过错缺陷和遗漏缺陷。如果把某些信息输入到不正确的表示中，就是过错缺陷；如果没有输入正确信息，就是遗漏缺陷。在这两类缺陷中，遗漏缺陷更难检测和解决。

失效（failure）——当缺陷执行时会发生失效。有两点需要解释：一是失效只出现在可执行的表现中，通常是源代码，或更确切地说是被装载的目标代码；二是这种定义只与过错缺陷有关。我们如何处理遗漏缺陷对应的失效呢？把这个问题再向前推进一步：应该怎样处理在执行中从来不发生，或可能在相当长时间内没有发生的缺陷呢？米开朗基罗（Michaelangelo）病毒就是这种缺陷的一个例子。这种病毒只有到米开朗基罗3月6日的生日那天才会发作。评审可以通过发现缺陷避免很多失效的发生。事实上，有效的评审可以找出遗漏缺陷。

事故（incident）——当出现失效时，可能会也可能不会呈现给用户（或客户或测试人员）。事故说明出现了与失效类似的情况，警告用户注意所出现的失效。

测试（test）——测试显然要处理错误、缺陷、失效和事故。测试是采用测试用例执行软件的活动。测试有两个显著目标：找出失效，或演示正确的执行。

测试用例（test case）——测试用例有一个标识，并与程序行为有关。测试用例还有一组输入和一个预期输出表。

图1-1给出了测试生命周期模型。请注意，在开发阶段，有三次机会可能引入错误，导致产生通过在开发过程的其余部分传播的缺陷。一位杰出的测试人员将这种生命周期归纳为：前三个阶段是“引入程序错误”，测试阶段是“找出程序错误”，后三个阶段是“清除程序错误”（Poston，1990）。“缺陷解决”步骤是另一个引入错误（以及新缺陷）的机会。当修复导致以前正确的软件出现错误行为时，这种修复就是不完善的。本书在讨论回归测试时还会讨论这个问题。

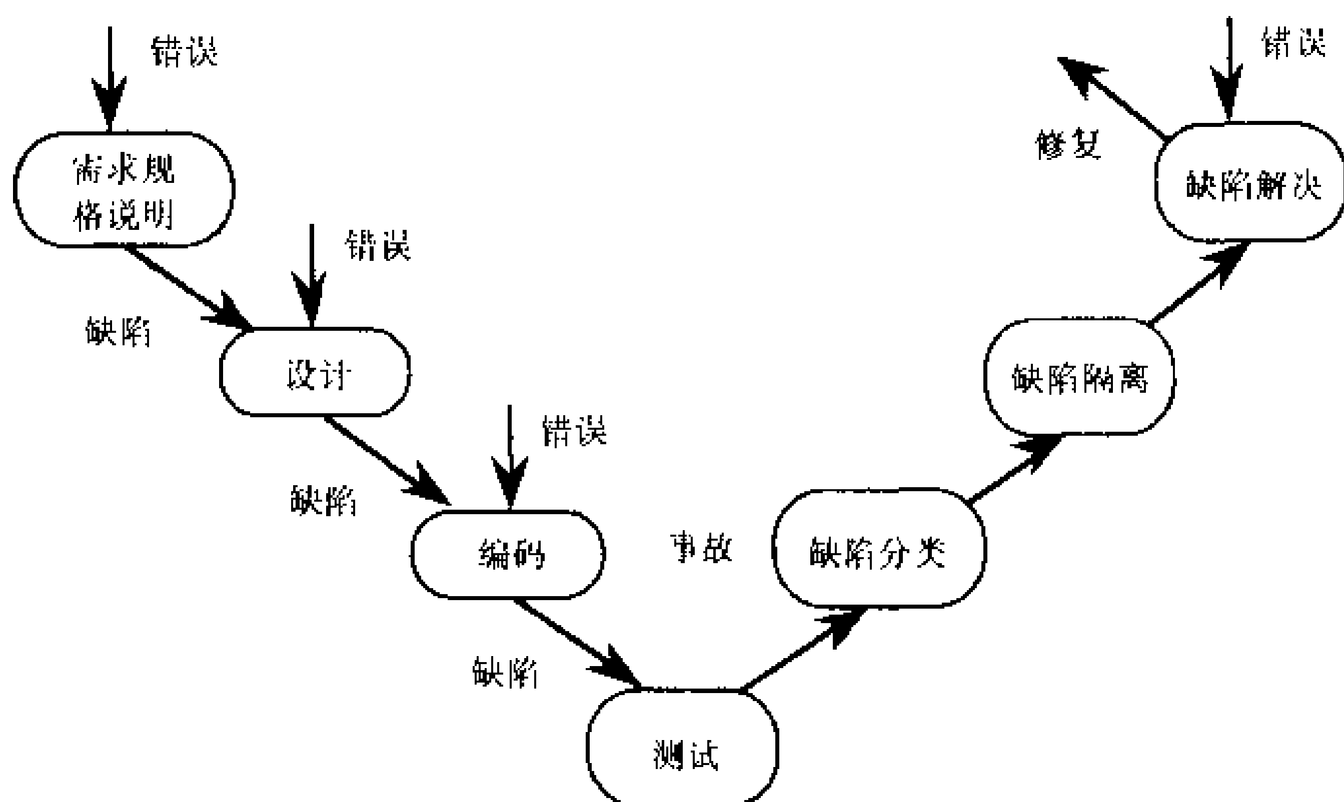


图1-1 一个测试生命周期

通过这些术语，可以发现测试用例在测试中占据中心地位。测试过程可以再细分为独立的步骤：测试计划、测试用例开发、运行测试用例以及评估测试结果。本书的重点是如何标识有用的测试用例集合。

1.2 测试用例

软件测试的本质是针对要测试的内容确定一组测试用例。在继续讨论之前，我们需要首先说明测试用例应该包含什么信息：

输入实际上有两种类型：前提（在测试用例执行之前已经存在的环境）和由某种测试方法所标识的实际输入。

预期输出也有两类：后果和实际输出。

测试用例的输出部分常常被忽视，这很不幸，因为输出部分的确定往往很困难。例如，假设要测试一个为飞机确定最佳航线的软件，要有一定的联邦航空管理局空中走廊约束条

件和飞行当天的气象数据。怎样才能知道实际最佳航线是什么呢？有各种各样的回答可以解决这个问题。学术界的回答是假设有一位“知道所有答案”的圣人。工业界的一种回答叫做“参考测试”（Reference Testing），即系统要在专家用户的指导下进行测试。这些专家要判断被执行的一组测试用例的输出是否为可接受的。

测试活动要建立必要的前提条件，提供测试用例输入，观察输出，然后将这些输出与预期输出进行比较，以确定该测试是否通过。

开发良好的测试用例的其他信息（如图1-2所示）主要支持测试管理。测试用例应该拥有一个标识和一个原因（需求跟踪是一个很好的原因）。记录测试用例的执行历史也是很有用的，包括测试用例是什么时候由谁运行的，每次执行的通过/失败记录，测试用例测试的是（软件的）哪个版本。清楚地给出这些信息会使测试用例更有价值——至少与源代码一样有价值。测试用例需要被开发、评审、使用、管理和保存。

测试用例ID
目的
前提
输入
预期输出
后果
执行历史
日期　结果　版本　执行人

图1-2 典型的测试用例信息

1.3 通过维恩图理解测试

测试基本上关心的是行为，而行为与软件（和系统）开发人员很常见的结构视图无关。最明显的差别是，结构视图关注的是它是什么，而行为视图关注的是它做什么。一直困扰测试人员的难点之一，就是基本文档通常都是由开发人员编写，并且是针对开发人员的，因此这些文档强调的是结构信息，而不是行为信息。本节要开发一种简单的维恩图，以澄清有关测试的几个争论不休的问题。

考虑一个程序行为全域。（请注意，我们关注的是测试的本质问题。）给定一段程序及其规格说明，集合S是所描述的行为，集合P是用程序实现的行为。图1-3给出了我们的论域以及所描述和用程序实现的行为之间的关系。对于所有可能的程序行为，描述行为都位于标有S的圆圈内，所有实际的程序行为都位于标有P的圆圈内（请注意P、U和全域之间的细微差别）。通过这张图，我们可以更清晰地看出测试人员所面临的问题。如果特定的描述行为

没有被编程实现会出现什么问题？用本章前面定义的术语说，这就是遗漏缺陷。类似地，如果特定的程序（已实现）行为没有被描述会出现什么问题？这种情况对应过错缺陷，以及规格说明完成之后出现的错误。S和P相交的部分（橄榄球型区域）是"正确"部分，即既被描述又被实现的行为。测试的一种很好的观点是，测试就是确定既被描述又被实现的程序行为的范围。（顺便说一下，请注意"正确性"只有在一个规格说明和一种实现背景下才有意义，正确性是一种相对术语，不是绝对术语。）

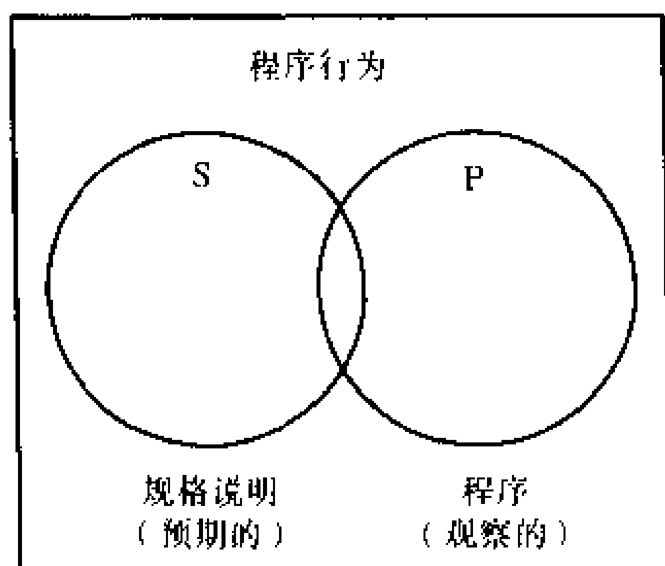


图1-3 所描述的行为与所实现的程序行为

在图1-4中新增加的圆圈代表的是测试用例。请注意我们的论域与程序行为集合之间的细微差别。由于一个测试用例要产生一个程序行为，因此数学家会谅解我们。现在考虑集合S、P和T之间的关系。可能会有没有测试的已描述行为（区域2和5）、经过测试的已描述行为（区域1和4），以及对应于未描述行为的测试用例（区域3和7）。

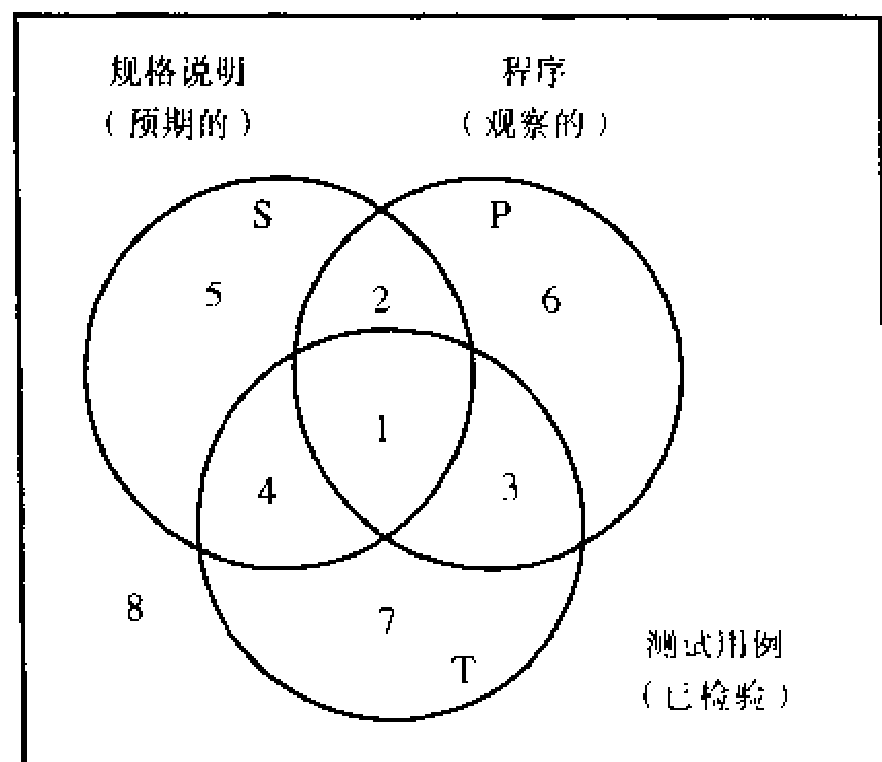


图1-4 已描述、已实现和经过测试的行为

类似地，也可能会有没有测试的程序行为（区域2和6），经过测试的程序行为（区域1和3），以及对应于未通过程序实现的行为（区域4和7）。这些区域中的每一个都很重要。如果测试用例没有对应的已描述行为，则测试一定是不完备的。如果特定测试用例对应未描述行为，则有两种可能：要么这个测试用例是不正当的，要么规格说明是不充分的。（根据我的经验，优秀测试人员经常会产生后一种类型的测试用例。这就是要请优秀测试人员参加规格说明和设计评审的一个很好理由。）

这时我们已经能够看出测试作为一种工艺的一些可能性：测试人员怎样才能使这些集合都相交的区域（区域1）尽可能地大？另一种方法是问如何标识集合T中的测试用例。这个问题的简单回答是，采用某种测试方法标识测试用例。这种框架提供了一种比较不同测试方法有效性的途径，第8章和第11章还将讨论这个问题。

1.4 标识测试用例

有两种基本方法可以用来标识测试用例，即功能性测试和结构性测试。这两种方法都有一些不同的测试用例标识方法，常常叫做测试方法。

1.4.1 功能性测试

功能性测试的基本观点是，任何程序都可以看作是将从输入定义域取值映射到输出值域的函数。（函数、定义域和值域都将在第3章中定义。）这种观点常常在工程中使用，将系统看作是黑盒。于是产生术语黑盒测试，其中，黑盒的内容（实现）是不知道的，而用输入和输出表示的黑盒函数则被完全了解（如图1-5所示）。在《摩托车维护的技巧与艺术》（Zen and the Art of Motorcycle Maintenance）中，Pirsig把它叫做“浪漫”理解（Pirsig，1973）。很多时候我们可以运用黑盒知识很有效地操作。事实上，这种方法是面向对象的核心。例如，大多数人都可以成功地仅仅凭借黑盒知识来操作摩托车。

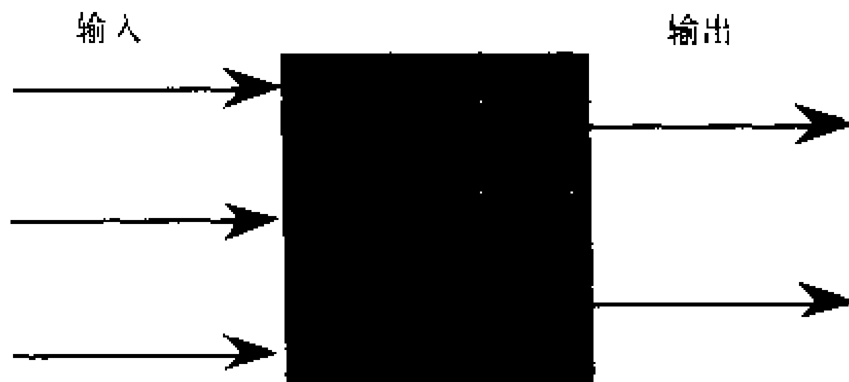


图1-5 工程师的黑盒

采用功能性方法标识测试用例，所使用的惟一信息就是软件的规格说明。功能性测试用例具有两个显著的优点：（1）功能性测试与软件如何实现无关，所以如果实现发生变化，测试用例仍然有用；（2）测试用例开发可以与实现并行进行，因此可压缩总的项目开发时

间。在缺点方面，功能性测试用例也常常会带来两个问题：测试用例之间可能存在严重的冗余，此外可能还会有未测试的软件漏洞。

图1-6给出了由两种功能性方法标识测试用例得到的结果。方法A标识了比方法B更大的测试用例集合。请注意，对于这两种方法，测试用例集合完全局限在已描述行为集合内。由于功能性方法基于已描述行为，因此很难想像这些方法能够标识没有被描述的行为。第8章将根据第2章定义的例子，直接比较由各种功能性方法生成的测试用例。

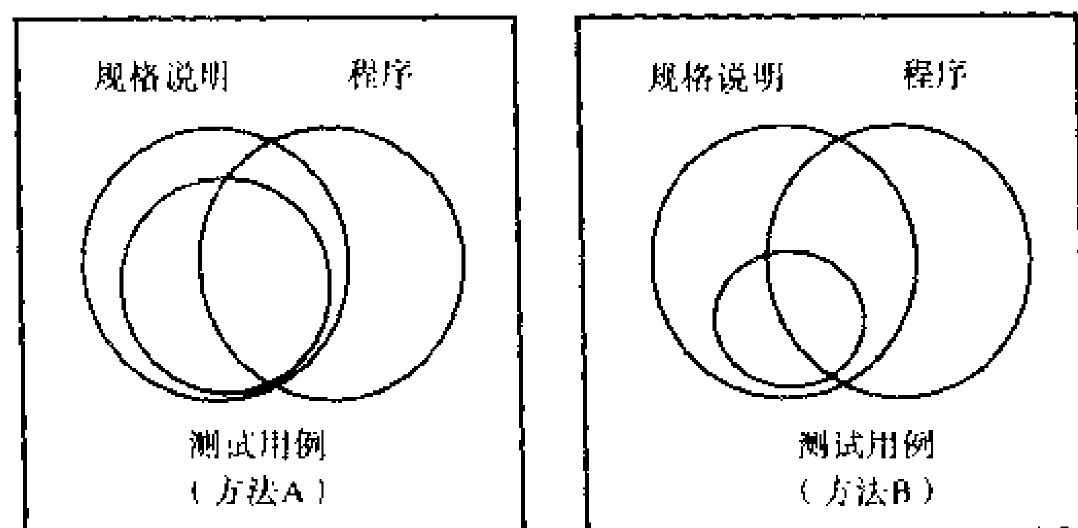


图1-6 功能性测试用例标识方法比较

第二部分将讨论功能性测试的主流方法，包括边界值分析、健壮性分析、最坏情况分析、特殊值测试、输入（定义域）等价类、输出（值域）等价类和基于决策树的测试。这些手段的共同特征是，都基于被测软件的定义信息。第3章将介绍的数学背景知识，主要适用于功能性方法。

1.4.2 结构性测试

结构性测试是另一种用于标识测试用例的基本方法。为了与功能性测试形成对比，结构性测试有时叫做白盒（或甚至叫做透明盒）测试。透明盒的比喻可能更恰当，因为根本差别在于（黑盒的）实现是已知的，并被用来标识测试用例。“看到黑盒内部”的能力，使测试人员能够根据功能实际实现的方式来标识测试用例。

结构性测试一直是一些相当强的理论的主题。为了真正理解结构性测试，熟悉线性图论的概念是很关键的（请参阅第4章）。通过这些概念，测试人员可以严格描述要测试的确切内容。由于具有很强的理论基础，结构性测试本身又引出测试覆盖指标的定义和使用。测试覆盖指标提供明确描述软件项测试范围的方法，而这又使测试管理变得更有意义。

图1-7给出了由两个结构性方法标识的测试用例结果。与图1-6一样，方法A也标识了比方法B更大的测试用例集合。更大的测试用例集合就一定更好吗？这是一个很好的问题，而结构性测试提供了得到答案的主要方法。请注意，对两种方法，测试用例集合都完全局限于已编程实现的行为集合中。由于结构性方法依靠的是程序，因此很难想像这些方法能够标识没有编程实现的行为。但是，不难想像结构性测试用例集合相对编程实现行为全集更

小。到第三部分结束时，我们就会看到由不同结构性方法生成的测试用例的直接比较

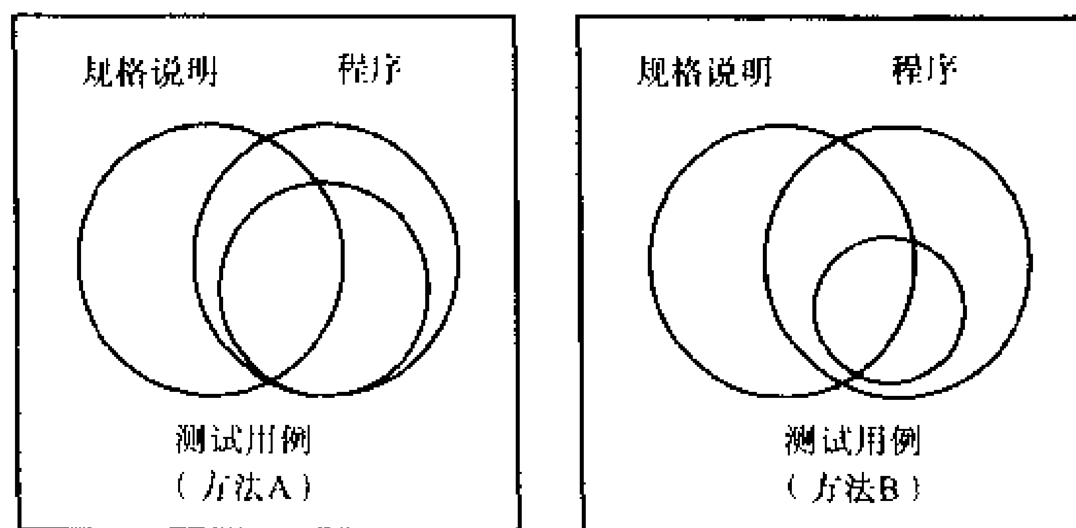


图1-7 结构性测试用例标识方法比较

1.4.3 功能性测试与结构性测试的比较

对于这两种根本不同的测试用例标识方法，很自然会产生哪种方法更好的问题。如果读者阅读过很多文献，就会发现两种方法都有坚定的拥护者。对于结构性测试，Robert Poston写道："这种工具自20世纪70年代以来一直在浪费测试人员的时间……[它]不支持好的软件测试实践，应该从测试人员的工具包中剔除"（Poston，1991）。Edward Miller在为结构性测试辩护时写道："如果达到85%或更好的水平，分支覆盖率[一种结构性测试覆盖率指标]标识出的缺陷，一般是靠'直觉'[功能性]测试找出的缺陷的两倍"（Miller，1991）。

前面给出的维恩图为这种争论提供了很强的评判手段。前面已经讨论过，两种方法的目标都是标识测试用例。功能性测试只利用规格说明标识测试用例，而结构性测试使用程序源代码（实现）作为测试用例标识的基础。前面的讨论我们得出的结论是，两种方法单独使用都是不充分的。考虑程序行为：如果所有已描述行为都没有被实现，则结构性测试永远也不会认识到这一点。反过来，如果程序实现了没有被描述的行为，功能性测试用例永远也不会揭示这一点。（病毒是这种未描述行为的很好的例子。）很容易得出的答案是，两种方法都需要。测试工艺师的答案是，明智的组合会带来功能性测试的置信，以及结构性测试的度量。前面我们曾经提到，功能性测试常常会有冗余和漏洞两方面的问题。如果功能性测试结合结构性测试覆盖率指标执行，则这两个问题都会被发现并解决（如图1-8所示）。

测试的维恩图观点提供了一种最终理解。测试用例集合T和已描述行为、已实现行为的集合S、集合P之间有什么关系？显然，T中的测试用例由所使用的测试用例标识方法确定。要问的一个很好的问题是，这种方法有多合适（有效）？为了通过以上讨论得出一种结论，可回忆前面提到过的从错误到缺陷、失效和事故的因果关系链。如果知道容易犯什么错误，并且知道在被测软件中可能存在什么类型的缺陷，就可以利用这种知识运用更恰当的测试用例标识方法，而正是这一点使测试真正成为一种工艺。

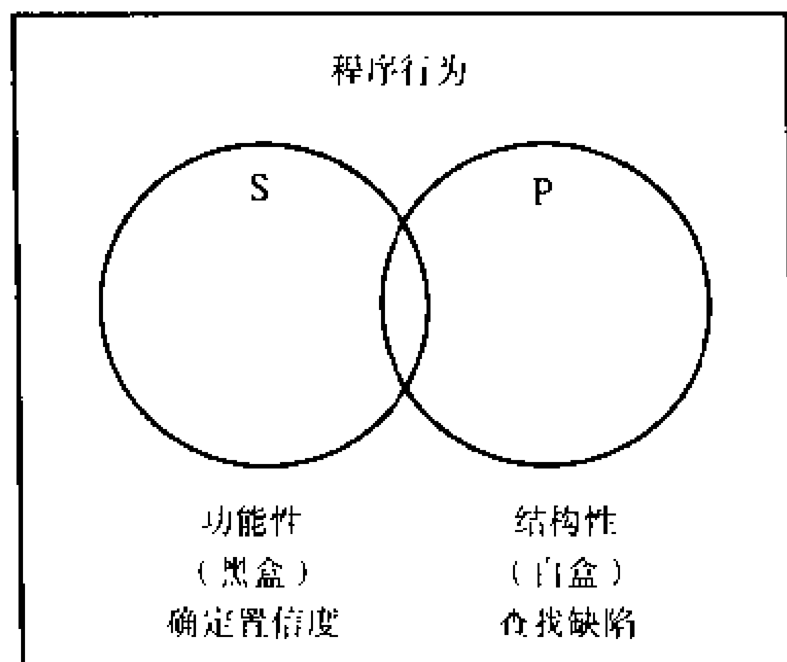


图1-8 测试用例来源

1.5 错误与缺陷分类

我们对错误和缺陷的定义，与过程和产品之间的差别有关：过程指我们怎样做事情，产品是过程的最终结果。测试与软件质量保证（SQA）的交汇点是SQA一般通过努力改进过程来改进产品。从这个意义上说，测试显然更面向产品。SQA更注重减少开发过程中的错误做法，而测试更注重发现产品中的缺陷。两种学科都会从缺陷类型的清晰定义获益。有多种方法可以对缺陷分类：以出现相应错误的开发阶段来划分，以相应失效产生的后果来划分，以解决难度来划分，以不解决会产生的风险来划分，等等。我更喜欢根据异常的出现来划分：只出现一次，间歇出现，反复出现或可重复出现。图1-9给出的是根据缺陷后果的严重程度来区分缺陷的一种缺陷分类（Beizer，1984）。

1. 轻微	词语拼写错误
2. 中等	误导或重复信息
3. 使人不悦	被截断的名称，0.00美元账单
4. 影响使用	有些交易没有处理
5. 严重	丢失交易
6. 非常严重	不正确的交易处理
7. 极为严重	经常出现“非常严重的”错误
8. 无法忍受	数据库破坏
9. 灾难性	系统停机
10. 容易传染	扩展到其他系统的系统停机

图1-9 根据严重程度分类的缺陷

有关缺陷类型的综合讨论，请参阅“软件异常IEEE标准分类”（IEEE，1993）。（在这份文件中，软件异常被定义为“对预期的偏离”，这与我们的定义很接近。）这个IEEE标准定义了围绕四个阶段（另一种生命周期）构建的详细的异常解决过程：识别、调查、行动和处置。表1-1～表1-5给出了一些更有用的异常，这些异常的大部分都摘自IEEE标准，不过也增加了一些我认为有用的异常。

表1-1　输入/输出缺陷

类型	举　例
输入	不接受正确的输入
	接受不正确的输入
	描述有错或遗漏
	参数有错或遗漏
输出	格式有错
	结果有错
	在错误的时间产生正确的结果（太早、太迟）
	不一致或遗漏结果
	不合逻辑的结果
	拼写/语法错误
	修饰词错误

表1-2　逻辑缺陷

遗漏情况
重复情况
极端条件出错
解释有错
遗漏条件
外部条件有错
错误变量的测试
不正确的循环迭代
错误的操作符（例如用<取代了≤）

表1-3　计算缺陷

不正确的算法
遗漏计算
不正确的操作数
不正确的操作
括号错误
精度不够（四舍五入，截断）
错误的内置函数

表1-4 接口缺陷

不正确的中断处理
I/O时序有错
调用了错误的过程
调用了不存在的过程
参数不匹配（类型，个数）
不兼容的类型
过量的包含

表1-5 数据缺陷

不正确的初始化
不正确的存储/访问
错误的标志/索引值
不正确的打包/拆包
使用了错误的变量
错误的数据引用
缩放数据范围或单位错误
不正确的数据维数
不正确的下标
不正确的类型
不正确的数据范围
传感器数据超出限制
出现1次断开
不一致的数据

1.6 测试级别

到目前为止，我们还没有提到测试的一个关键概念，即抽象级别。测试级别反映软件开发生命周期瀑布模型中的抽象级别。虽然这种模型有缺点，但是它作为一种标识不同的测试级别，以及澄清适合每个测试级别的目标的手段，非常有用。瀑布模型框图的一个变种如图1-10所示，这个变种强调测试与设计级别的对应关系。请注意，尤其是在功能性测试方面，有三个级别的定义（规格说明、概要设计、详细设计）直接对应于测试的三个级别，即单元测试、集成测试和系统测试。

测试级别与功能性和结构性测试存在现实的关系。大多数实践者都同意结构性测试最适合在单元级别上进行，而结构性测试最适合在系统级别上进行。这在一般情况下是正确的，但是也有可能按照需求规格说明、概要设计和详细设计阶段产生的基本信息的顺序。结构性测试定义的构造在单元级别上最有意义，集成和系统级别的测试只是到现在才有类似的构造。我们将在第四部分开发这种结构，以支持传统软件和面向对象软件的集成和系统级别的结构性测试。

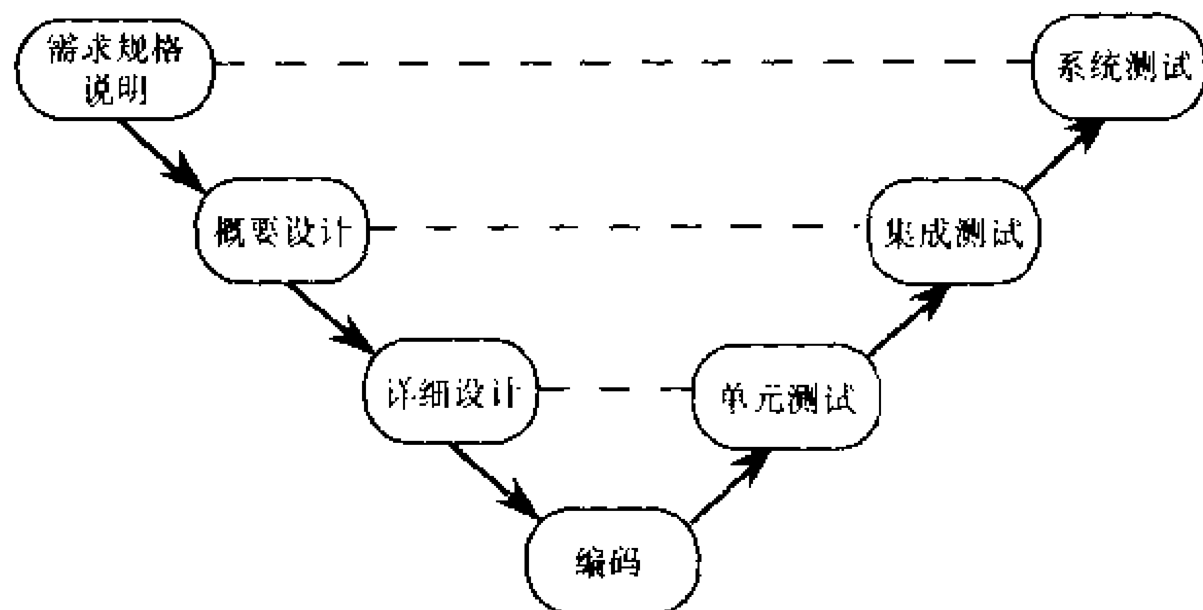


图1-10 瀑布模型中的抽象和测试级别

1.7 参考文献

Beizer, Boris, *Software System Testing and Quality Assurance*, Van Nostrand Reinhold, New York, 1984.

IEEE Computer Society, *IEEE Standard Glossary of Software Engineering Terminology*, 1983, ANSI/IEEE Std 729-1983.

IEEE Computer Society, *IEEE Standard Classification for Software Anomalies*, 1993, IEEE Std 1044-1993.

Miller, Edward F. Jr., Automated software testing. A technical perspective, *American Programmer*, April 1991, 4:4, 38–43.

Pirsig, Robert M., *Zen and the Art of Motorcycle Maintenance*, Bantam Books, New York, 1973.

Poston, Robert M., *T: Automated Software Testing Workshop*, Programming Environments, Inc., Tinton Falls, NJ, 1990.

Poston, Robert M., A complete toolkit for the software tester, *American Programmer*, April 1991, 4:4, 28–37. Reprinted in CrossTalk, a USAF publication.

1.8 练习

1. 画出反映以下陈述的维恩图："……我们没有完成本来应该完成的工作，我们完成了本来不应该完成的工作……"。

2. 描述图1-4中的八个区域。你可以根据自己编写的软件，举出属于这些区域的例子吗？

3. 有一个软件知识方面的故事，说的是一个令人不愉快的员工要编写一段工资管理程序。这段程序包含在生成工资支票之前检查员工标识编号的逻辑。如果该员工被终止在公司的工作，则该程序会产生严重破坏。采用错误、缺陷和失效模式讨论这个例子，并确定什么形式的测试最适合。

第2章
举　　例

第二和第三部分将通篇采用三个例子描述各种单元测试方法。这三个例子是：三角形问题（测试界的一个古老例子）；逻辑比较复杂的函数，NextDate；有代表性的MIS测试，这里叫做佣金问题。这三个例子合在一起，可说明测试工艺师在单元级别上会遇到的大多数问题。第四部分在讨论集成和系统测试时要使用另外三个例子：一个自动柜员机（ATM）的简化版本，这里叫做简单ATM系统（SATM）；货币转换器，一种事件驱动的应用程序，这是典型的图形用户界面（GUI）应用程序；土星汽车公司的挡风玻璃雨刷。最后，我们给出一个NextDate的面向对象版本，叫做o-oCalendar，用于在第五部分说明面向对象软件测试方面的问题。

对于结构性测试，本章给出三个单元级例子的伪代码实现。SATM系统、货币转换器和土星牌挡风玻璃雨刷系统的系统级描述，将在第四部分给出。在第五部分中，这些应用程序将既采用传统方式描述（通过E/R图、数据流图和有限状态机），也采用事实上的面向对象标准统一建模语言（UML）描述。

2.1　泛化的伪代码

泛化伪代码提供表示程序源代码的"独立于语言"的方式。伪代码有两层结构：单元和程序组件。单元既可以解释为传统组件（过程和函数），也可以解释为面向对象的组件（类和对象）。这种定义有些不太正式，像表达式、变量表和字段描述等术语，都不加正式定义地被使用。尖括号中的内容表示可以在所标识位置上使用的语言要素。所有伪代码的一部分价值都是为了去掉不想要的细节，这里我们通过允许在形式化的复杂条件中加入自然语言短语来说明这个问题（请参阅表2-1）。

表2-1　泛化的伪代码

语言要素	泛化的伪代码结构
注释	'<文本>
数据结构声明	Type<类型名称>
	<字段描述列表>
	End<类型名称>
数据声明	Dim<变量>As<类型>
赋值语句	<变量>=<表达式>

（续）

语言要素	泛化的伪代码结构
输入	Input(<变量列表>)
输出	Output(<变量列表>)
简单条件	<表达式><关系操作符><表达式>
复合条件	<简单条件><逻辑连接符><简单条件>
序列	语句按串行顺序排列
简单选择	If<条件>Then <then子句> EndIf
选择	If<条件>Then <then子句> Else<else子句> EndIf
多重选择	Case<变量>Of Case 1:<谓词> <case子句> ... Case n:<谓词> <case子句> EndCase
计数器控制的重复	For<计数器>=<开始>To<结束> <循环体> EndFor
预测试重复	Do While<条件> <循环体> EndWhile
后测试重复	Do <循环体> Until<条件>
过程定义（函数和面向对象方法的定义类似）	<过程名称>(Input:<变量列表>; Output:<变量列表>) <主体> End<过程名称>
单元间通信	Call<过程名称>(<变量列表>; <变量列表>)
类/对象定义	<名称>(<属性列表>; <方法列表>，<主体>) End<名称>
单元间通信	Msg<目的对象名称>.<方法名称>(<变量列表>)
对象创建	Instantiate<类名称>.<对象名称>(<属性值>)
对象销毁	Delete<类名称>.<对象名称>
程序	Program<程序名称> <单元列表> End<程序名称>

2.2 三角形问题

三角形问题是在软件测试文献中使用最广的一个例子，其中30年来更引人注意的测试文献是Gruenberger（1973）、Brown（1975）、Myers（1979）、Pressman（1982）及其第2、3、4和5版、Clarke（1983）、Clarke（1984）、Chellappa（1987）和Hetzel（1988）。当然还有其他很多，不过这些已经可以说明问题了。

2.2.1 问题陈述

简单版本：三角形问题接受三个整数a、b和c作为输入，用做三角形的边。程序的输出是由这三条边确定的三角形类型：等边三角形、等腰三角形、不等边三角形或非三角形。有时这个问题被扩展为将直角三角形作为第五类，在有些练习中会使用这种扩展。

通过提供更多细节可以改进这个定义。于是这个问题变成以下的形式。

改进版本：三角形问题接受三个整数a、b和c作为输入，用做三角形的边。整数a、b和c必须满足以下条件：

c1. $1 \leq a \leq 200$	c4. $a < b + c$
c2. $1 \leq b \leq 200$	c5. $b < a + c$
c3. $1 \leq c \leq 200$	c6. $c < a + b$

程序的输出是由这三条边确定的三角形类型：等边三角形、等腰三角形、不等边三角形或非三角形。如果输入值没有满足这些条件中的任何一个，则程序会通过输出消息来进行通知，例如，“b的取值不在允许取值的范围内。”如果a、b和c取值满足c1、c2和c3，则给出以下四种相互排斥输出中的一个：

1. 如果三条边相等，则程序的输出是等边三角形。
2. 如果恰好有两条边相等，则程序的输出是等腰三角形。
3. 如果没有两条边相等，则程序输出的是不等边三角形。
4. 如果c4、c5和c6中有一个条件不满足，则程序输出的是非三角形。

2.2.2 讨论

这个例子经久不衰的原因之一是，它包含了清晰而又复杂的逻辑。它还是削弱客户、开发人员和测试人员之间沟通的不完整定义的典型例子。第一个规格说明假设开发人员知道有关三角形的一些细节，特别是三角形性质：任何两边的和必须严格大于第三条边。上限200既是任意的也是方便的，第5章在开发边界值测试用例时要使用这个上限值。

2.2.3 传统实现

这个所有例子老祖先的“传统”实现，具有类似FORTRAN的风格。图2-1给出了这个实

现的流程图. 流程图中的框号与后面（类似FORTRAN的）伪代码程序中的注释号对应。（这些编号与Pressman[1982]中的编号完全对应）。我不太喜欢这种实现，因此在第2.2.4节将给出一种更结构化的实现。

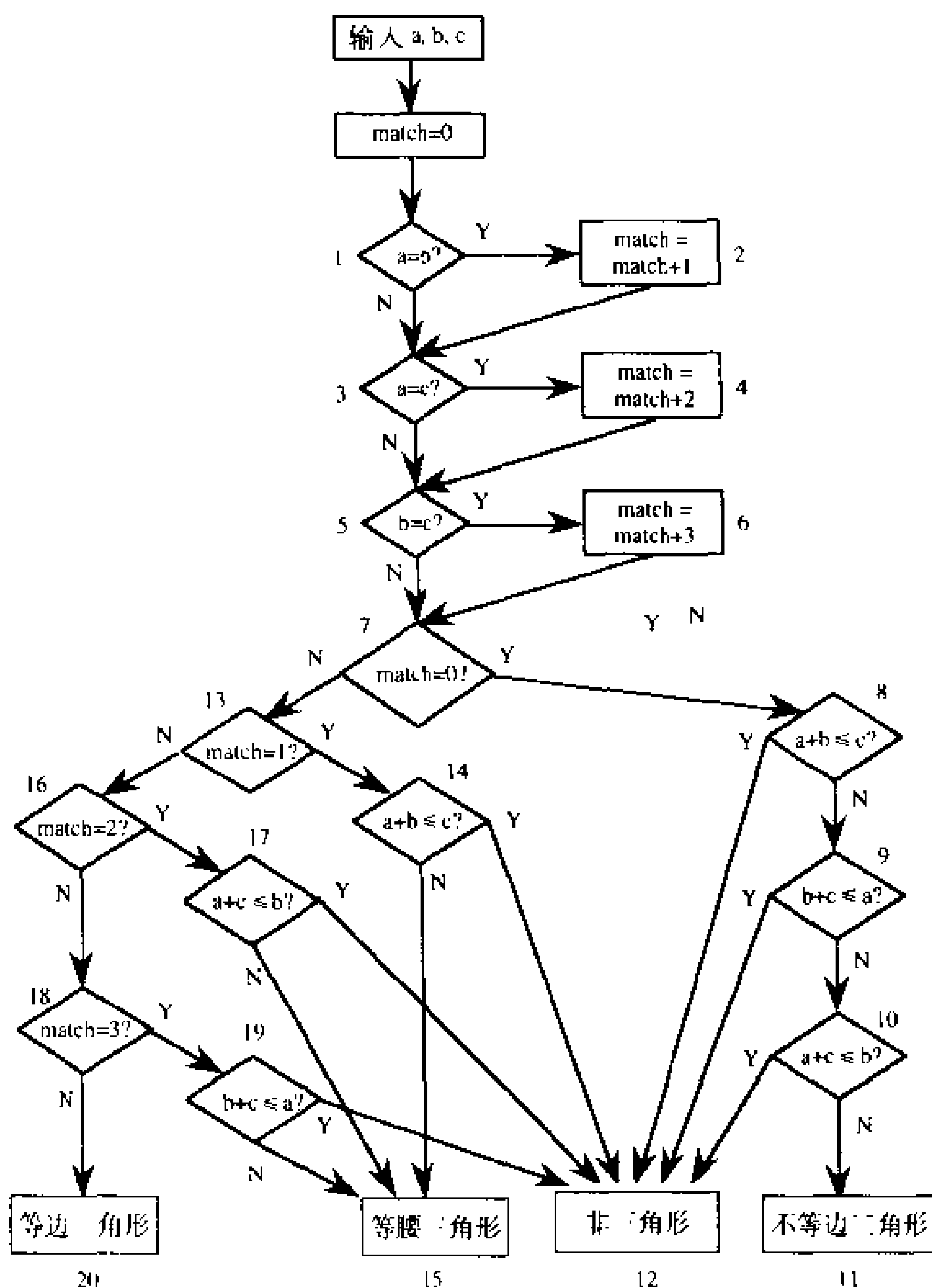


图2-1 传统三角形问题实现的流程图

变量match用来记录各对边的相等情况。经典FORTRAN风格之所以难以理解，有一点是与变量match有关的：请注意，三角形性质的三个测试都没有发生。如果两条边相等，例如a和c，那么只需要将a + c与b比较。（因为b一定大于0，a + b一定大于c，因为c等于a。）这种观察显然可以减少必须进行的比较次数。这个版本的效率是以牺牲清晰性（和易测性）为代价换来的。当第三部分讨论不可行的程序执行路径时，可以看出这个版本是很有用的。这是保留这个版本的惟一原因。

```
Program triangle1 'Fortran-like version
'
Dim a,b,c,match As INTEGER
'
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
match = 0
If a = b                                                          '(1)
   Then match = match + 1                                         '(2)
EndIf
If a = c                                                          '(3)
   Then match = match + 2                                         '(4)
EndIf
If b = c                                                          '(5)
   Then match = match + 3                                         '(6)
EndIf
If match = 0                                                      '(7)
   Then  If (a+b)<=c                                              '(8)
            Then  Output("NotATriangle")                          '(12.1)
            Else  If (b+c)<=a                                     '(9)
                     Then  Output("NotATriangle")                 '(12.2)
                     Else  If (a+c)<=b                            '(10)
                              Then  Output("NotATriangle")        '(12.3)
                              Else  Output ("Scalene")            '(11)
                           EndIf
                  EndIf
         EndIf
   Else  If match=1                                               '(13)
            Then  If (a+c)<=b                                     '(14)
                     Then  Output("NotATriangle")                 '(12.4)
                     Else  Output ("Isosceles")                   '(15.1)
                  EndIf
            Else  If match=2                                      '(16)
                     Then  If (a+c)<=b
                              Then  Output("NotATriangle")        '(12.5)
                              Else  Output ("Isosceles")          '(15.2)
                           EndIf
                     Else  If match=3                             '(18)
                              Then  If (b+c)<=a                   '(19)
                                       Then  Output("NotATriangle") '(12.6)
                                       Else  Output ("Isosceles")   '(15.3)
                                    EndIf
                              Else  Output ("Equilateral")        '(20)
                           EndIf
                  EndIf
         EndIf
EndIf
'
End Triangle1
```

请注意，有六个途径可用来到达非三角形方框（12.1~12.6），有三个途径可用来到达等腰三角形方框（15.1~15.3）。

2.2.4 结构化实现

图2-2是三角形程序的数据流图描述。可以把这个程序实现为一个主程序和三个简单的过程。稍后在讨论单元测试时还要使用这个例子，因此将三个过程合并为一个伪代码程序，注释行与图2-2给出的分解代码相关部分关联。

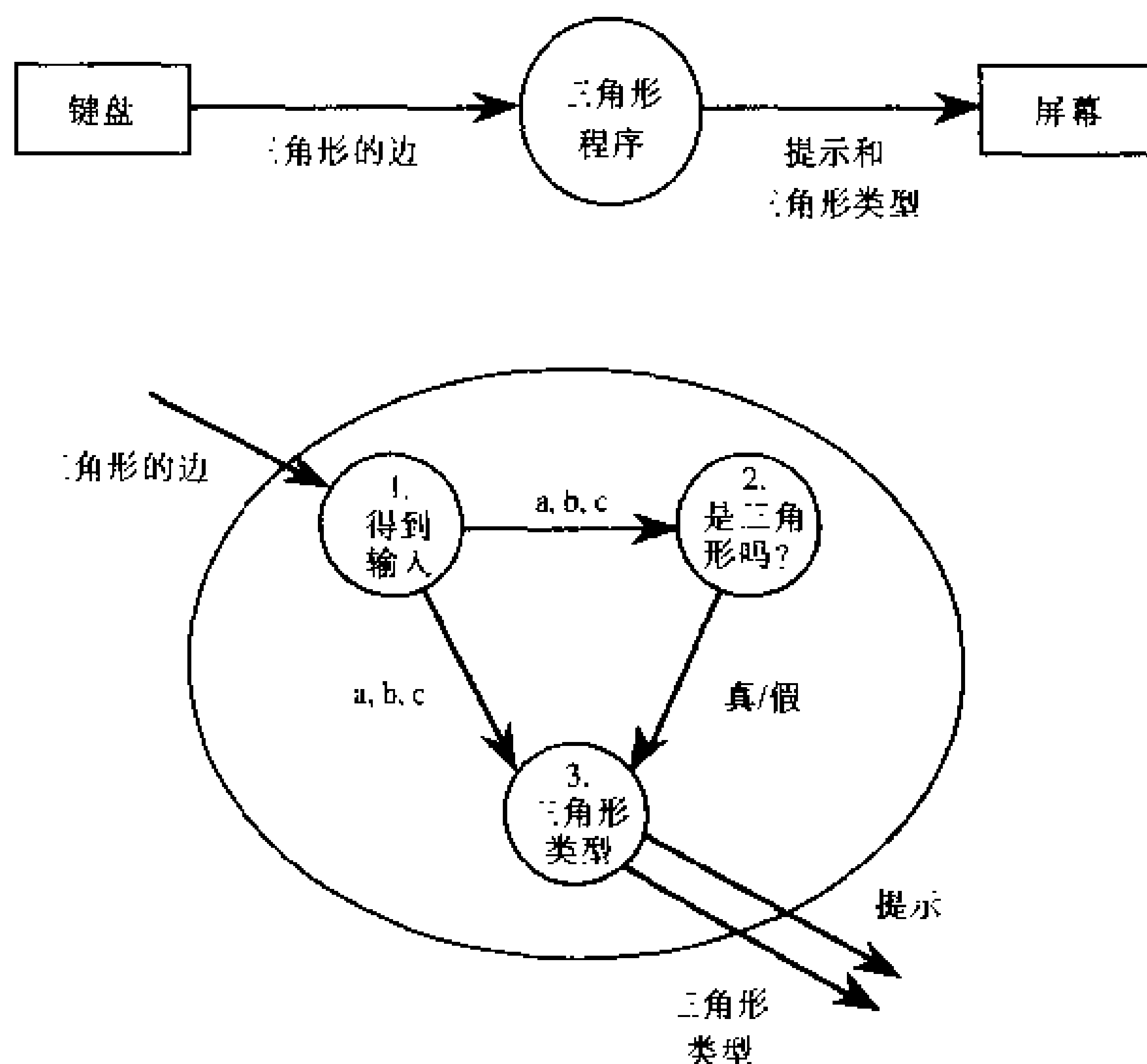


图2-2 结构化三角形程序实现的数据流图

```
Program triangle2 'Structured programming version of simpler specification
'
Dim a,b,c As Integer
Dim IsATriangle As Boolean
'
'Step 1: Get Input
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
'
'Step 2: Is A Triangle?
If (a < b + c) AND (b < a + c) AND (c < a + b)
    Then IsATriangle = True
    Else IsATriangle = False
EndIf
'
'Step 3: Determine Triangle Type
If IsATriangle
```

```
   Then    If (a = b) AND (b = c)
              Then Output ("Equilateral")
              Else    If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                          Then    Output ("Scalene")
                          Else    Output ("Isosceles")
                      EndIf
           EndIf
   Else    Output("Not a Triangle")
EndIf
'
End triangle2


Program triangle3 'Structured programming version of improved specification
'
Dim a,b,c As Integer
Dim c1, c2, c3, IsATriangle As Boolean
'
'Step 1: Get Input
Do
    Output("Enter 3 integers which are sides of a triangle")
    Input(a,b,c)
    c1 = (1 <= a) AND (a <= 200)
    c2 = (1 <= b) AND (b <= 200)
    c3 = (1 <= c) AND (c <= 200)
    If NOT(c1)
        Then    Output("Value of a is not in the range of permitted values")
    EndIf
    If NOT(c2)
        Then    Output("Value of b is not in the range of permitted values")
    EndIf
    If NOT(c3)
        Then    Output("Value of c is not in the range of permitted values")
    EndIf
Until c1 AND c2 AND c3
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
'
'Step 2: Is A Triangle?
If (a < (b + c)) AND (b < (a + c)) AND (c < (a + b))
    Then IsATriangle = True
    Else IsATriangle = False
EndIf
'
'Step 3: Determine Triangle Type
If IsATriangle
   Then    If (a = b) AND (b = c)
              Then Output ("Equilateral")
              Else    If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                          Then    Output ("Scalene")
                          Else    Output ("Isosceles")
                      EndIf
           EndIf
   Else    Output("Not a Triangle")
EndIf
'
End triangle3
```

2.3 NextDate函数

三角形程序之所以复杂，是因为输入与正确输出之间的关系复杂。以下利用NextDate函数说明另一种复杂性，即输入变量之间的逻辑关系复杂性。

2.3.1 问题陈述

NextDate是一个有三个变量（月份、日期和年）的函数。函数返回输入日期后面的那个日期。变量月份、日期和年都具有整数值，且满足以下条件：

c1. 1≤月份≤12

c2. 1≤日期≤31

c3. 1812≤年≤2012

与处理三角形程序一样，我们也可以使规格说明更具体，包括对日期、月份和年的无效输入值的响应定义，还可以对无效逻辑组合定义，例如任意年的6月31日的响应。如果c1、c2或c3中的任意一个条件失败，则NextDate都会产生一个输出，指示相应的变量超出取值范围，例如，“月份值不在1~12范围内”。由于存在大量的无效日期-月份-年组合，因此NextDate将这些组合的消息合并为一个消息：“无效输入日期。”

2.3.2 讨论

在NextDate函数中有两种复杂性来源：以上所讨论的输入域的复杂性，以及确定哪一年是闰年的规则。一年有365.2422天，因此，闰年被用来解决“额外天”的问题。如果每四年定义一个闰年，则会出现小错误。罗马日历（Gregory教皇之后）通过调整一百整数年的闰年来解决这个问题。因此，如果年数可以被4整除则为闰年，除非是一百整数年。一百整数年是400的倍数时为闰年（Inglis，1961），因此1992、1996和2000年都是闰年，而1900不是闰年。NextDate函数还间接说明了软件测试。很多时候，我们可发现Zipf定律的例子，即80%的活动出现在20%的空间。请注意有多少行源程序是用来处理闰年的。在第二种实现中，请注意有多少行代码用来处理输入值的有效性。

2.3.3 实现

```
Program NextDate1        'Simple version
'
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
'
Output ("Enter today's date in the form MM DD YYYY")
Input (month,day,year)
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
```

```
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = month - 1
    EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
    If day < 30
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = month + 1
    EndIf
Case 3: month Is 12: 'December
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = 1
            If year = 2012
                Then Output ("2012 is over")
                Else tomorrow year = year + 1
    EndIf
Case 4: month is 2: 'February
    If day < 28
        Then tomorrowDay = day + 1
        Else
            If day = 28
                Then
                    If (year is a leap year)
                        Then tomorrowDay = 29 'leap year
                        Else           'not a leap year
                            tomorrowDay = 1
                            tomorrowMonth = 3
                    EndIf
                Else    If day = 29
                            Then tomorrowDay = 1
                                tomorrowMonth = 3
                            Else    Output("Cannot have Feb ", day)
                        EndIf
            EndIf
    EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
'
End NextDate


Program NextDate2        Improved version
'
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Dim c1, c2, c3 As Boolean
'
Do
    Output ("Enter today's date in the form MM DD YYYY")
```

```
    Input (month,day,year)
    c1 = (1 <= day) AND (day <= 31)
    c2 = (1 <= month) AND (month <= 12)
    c3 = (1812 <= year) AND (year <= 2012)
    If NOT(c1)
        Then   Output("Value of day not in the range 1..31")
    EndIf
    If NOT(c2)
        Then   Output("Value of month not in the range 1..12")
    EndIf
    If NOT(c3)
        Then   Output("Value of year not in the range 1812..2012")
    EndIf
Until c1 AND c2 AND c2

Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = month + 1
    EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
    If day < 30
        Then tomorrowDay = day + 1
        Else
            If day = 30
                Then   tomorrowDay = 1
                           tomorrowMonth = month + 1
                Else   Output("Invalid Input Date")
            EndIf
    EndIf
Case 3: month Is 12: 'December ,
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = 1
            If year = 2012
                Then Output ("Invalid Input Date")
                Else tomorrow.year = year + 1
    EndIf
Case 4: month is 2: 'February
    If day < 28
        Then tomorrowDay = day + 1
        Else
            If day = 28
                Then
                    If (year is a leap year)
                        Then tomorrowDay = 29 'leap day
                        Else          'not a leap year
                            tomorrowDay = 1
                            tomorrowMonth = 3
                    EndIf
                Else
                    If day = 29
```

```
                    Then
                        If (year is a leap year)
                            Then  tomorrowDay = 1
                                  tomorrowMonth = 3
                            Else
                                If day > 29
                                    Then  Output("Invalid Input Date")
                                EndIf
                            EndIf
                    EndIf
            EndIf
    EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)

End NextDate2
```

2.4 佣金问题

第三个例子是个典型的商务计算例子，包含了计算和决策，因此引出有意思的测试问题。

2.4.1 问题陈述

前亚利桑那州境内的一位步枪销售商销售密苏里州制造商制造的步枪机（lock）、枪托（stock）和枪管（barrel）。枪机卖45美元，枪托卖30美元，枪管卖25美元。销售商每月至少要售出一支完整的步枪，且生产限额是大多数销售商在一个月内可销售70个枪机、80个枪托和90个枪管。每访问一个镇子之后，销售商都给密苏里州步枪制造商发出电报，说明在那个镇子中售出的枪机、枪托和枪管数量。到了月末，销售商要发出一封很短的电报，通知－1个枪机被售出。这样步枪制造商就知道当月的销售情况，并计算销售商的佣金如下：销售额不到（含）1000美元的部分为10%，1000（不含）~1800（含）美元的部分为15%，超过1800美元的部分为20%。佣金程序生成月份销售报告，汇总售出的枪机、枪托和枪管总数、销售商的总销售额以及佣金。

2.4.2 讨论

这个例子有些人为地使读者能够快速地看到算法。考虑有多个变量的一些其他函数可能更现实，例如填写美国1040收入报税表的各种计算。（我们仍然讨论步枪。）这个问题分为三个不同的部分：输入数据部分，用来处理输入数据有效性（与在三角形和NextDate程序中做的一样）；销售额计算；以及佣金计算部分。这一次我们省略输入数据有效性部分，我们采用典型MIS数据采集应用程序所使用的哨兵控制While循环复述电报规则。

2.4.3 实现

```
Program Commission (INPUT,OUTPUT)
'
Dim locks, stocks, barrels As Integer
Dim lockPrice, stockPrice, barrelPrice As Real
Dim totalLocks,totalStocks,totalBarrels As Integer
Dim lockSales, stockSales, barrelSales As Real
Dim sales,commission : REAL
'
lockPrice  = 45.0
stockPrice  = 30.0
barrelPrice = 25.0
totalLocks = 0
totalStocks = 0
totalBarrels = 0
'
Input(locks)
While NOT(locks = -1)     'Input device uses -1 to indicate end of data
    Input(stocks, barrels)
    totalLocks = totalLocks + locks
    totalStocks = totalStocks + stocks
    totalBarrels = totalBarrels + barrels
    Input(locks)
EndWhile
'
Output("Locks sold: ", totalLocks)
Output("Stocks sold: ", totalStocks)
Output("Barrels sold: ", totalBarrels)
'
lockSales = lockPrice*totalLocks
stockSales = stockPrice*totalStocks
barrelSales = barrelPrice * totalBarrels
sales = lockSales + stockSales + barrelSales
Output("Total sales: ", sales)
'
If (sales > 1800.0)
    Then
        commission = 0.10 * 1000.0
        commission = commission + 0.15 * 800.0
        commission = commission + 0.20*(sales-1800.0)
    Else If (sales > 1000.0)
            Then
                commission = 0.10 * 1000.0
                commission = commission + 0.15*(sales-1000.0)
            Else commission = 0.10 * sales
        EndIf
EndIf
Output("Commission is $",commission)
'
End Commission
```

2.5 SATM系统

为了更好地讨论集成和系统测试问题，我们需要一个适用范围更大的例子。这里介绍的

自动柜员机是Topper（1993）的一种改进型，包含很有意思的各种功能和交互，是典型的客户-服务器系统的客户端。

2.5.1 问题陈述

SATM系统通过如图2-4所示的15个屏幕与银行客户通信。采用具有如图2-3所示功能的终端，SATM客户可以选择三种事务中的任意一种：存款、取款和余额查询。这些事务可以在两类账户上完成：支票账户和储蓄账户。

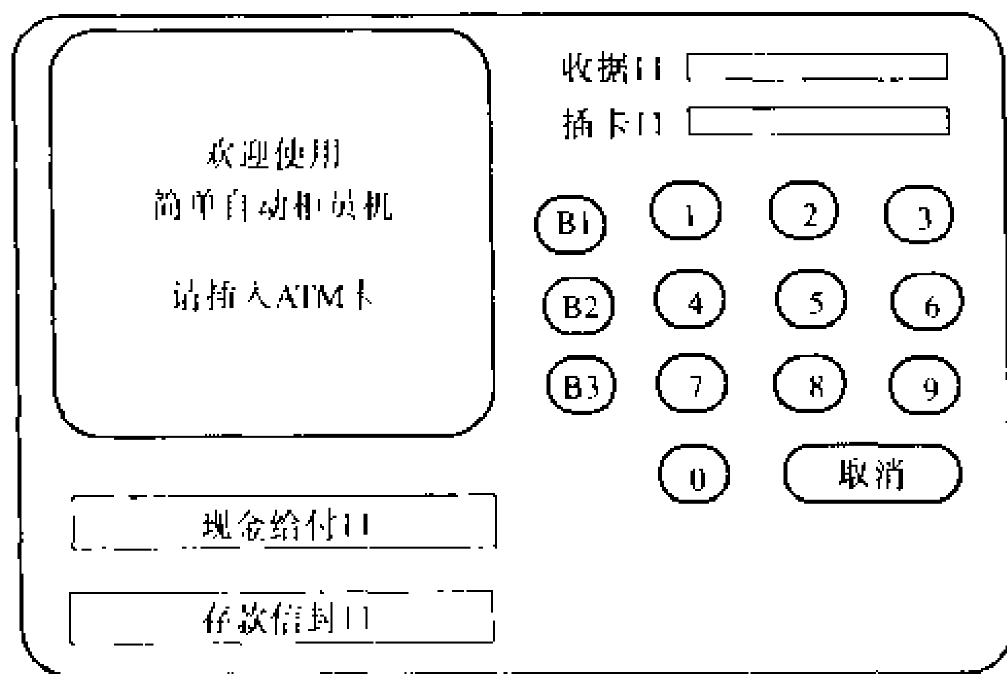


图2-3 SATM终端

当银行客户来到SATM机前时，显示屏幕1。银行客户通过带有个人账户编号（PAN）编码的塑料卡片访问SATM系统，这是打开包含客户姓名和账户信息等的内部客户账户文件的钥匙。如果客户的PAN与客户账户文件中的信息匹配，系统向客户显示屏幕2。如果没有找到客户的PAN，则显示屏幕4，并留下卡片。

当出现屏幕2时，系统会提示客户输入个人标识编号（PIN）。如果所输入的PIN正确（即与客户账户文件中的信息匹配），则系统显示屏幕5，否则显示屏幕3。客户有三次机会更正PIN，三次失败之后，会显示屏幕4，并留下卡片。

当出现屏幕5时，系统会向该客户的账户文件中增加两条信息：当前日期和ATM会话递进编号。客户从屏幕5中显示的选项中选择所需的事务，然后系统立即显示屏幕6，从中用户选择被选事务要处理的账户。

如果请求查询余额，则系统会检查本地ATM文件，寻找未完成的事务，并使这些事务与从该客户账户文件得到的该日开始余额一致。然后显示屏幕14。

如果请求存款，则存款信封槽的状态要通过终端控制文件中的一个字段确定。如果没有问题，则系统会显示屏幕7，获得事务金额。如果存款信封槽有问题，则系统显示屏幕12。一旦输入存款账户之后，系统显示屏幕13，接受存款信封，并处理存款。如果所输入的存款金额是本地ATM文件中的未完成金额，则每个月的存款计数加1。这些（以及其他信息）都由主ATM（中心）系统每天处理一次。然后系统显示屏幕14。



图2-4 SATM屏幕

如果请求取款，则系统检查终端控制文件中的取款通道状态（是堵塞还是可用）。如果堵塞，则显示屏幕10，否则显示屏幕7，使客户能够输入取款金额。一旦输入取款金额，系统要检查终端文件状态，查看是否有足够存款可供提取。如果存款不足，则显示屏幕9，否则处理取款。系统检查客户余额（与余额请求事务的处理过程相同），如果资金不足，则显示屏幕8。如果账户余额足够，则显示屏幕11并付现金。提取金额被写入未完成的本地ATM文件中，每月取款计数加1。余额打印在事务收据上，与余额请求事务的处理过程相同。客户取走现金之后，系统显示屏幕14。

如果客户按下屏幕10、12或14中的按钮"否"，则系统会显示屏幕15，并退回客户的ATM卡。一旦卡从卡槽中取走，系统会显示屏幕1。如果客户按下屏幕10、12或14中的按钮"是"，则系统显示屏幕5，使客户可以选择额外的事务。

2.5.2 讨论

有大量信息被“埋藏”在刚刚给出的系统描述中。例如，如果读者仔细地阅读，就可以发现该终端只有10美元面值的现金（请参阅屏幕7）。这种文字定义可能比实际通常遇到的情况更精确。这个例子有意进行了简化（因此取名SATM）。

其余问题可以通过一个假设列表解决。例如，是否有借款限制？如果客户使用多个ATM终端，怎样防止客户提取超过实际余额的现金？还有一些最初面临的问题：最初要放入机器中多少现金？怎样向系统中增加新客户？为了简化问题，这里不讨论这类现实世界的具体问题。

2.6 货币转换器

货币转换程序是另一种事件驱动的程序，强调与图形用户界面（GUI）关联的代码。图2-5给出了一个采用Visual Basic构建的GUI样例。

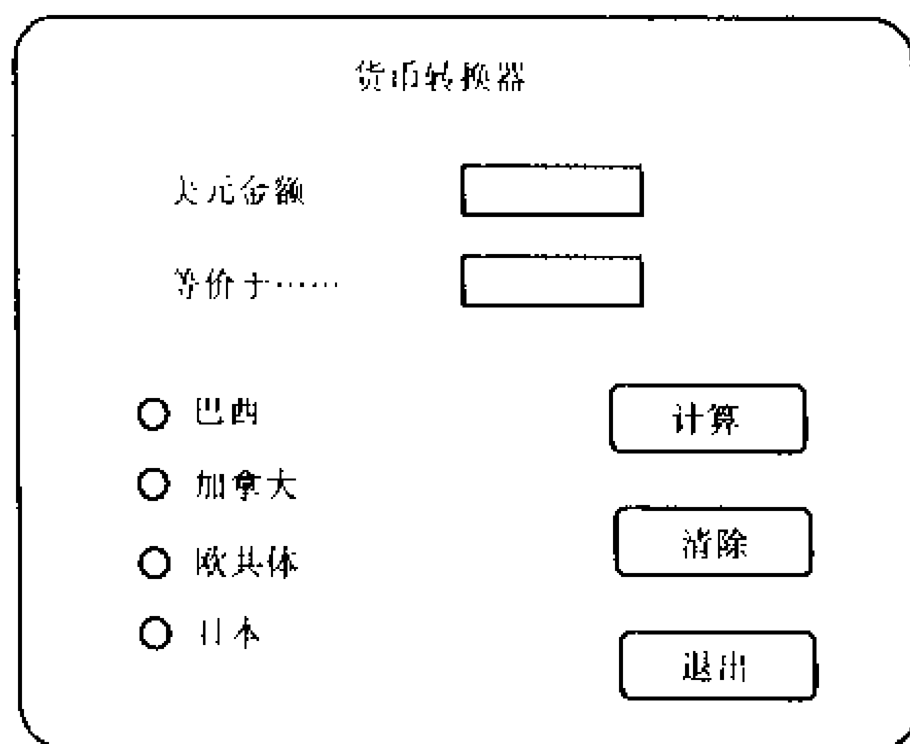


图2-5 货币转换器GUI

这个应用程序将美元转换为任意四种货币：巴西瑞尔、加拿大元、欧元和日元。货币选择由无线电按钮（Visual Basic选项按钮）控制，这些按钮相互排斥。当选择了一个国家时，系统会通过使标签句子变得完整来回应。例如，如果点击了“加拿大”按钮，则标签“等于……”会变成“等于加拿大元”。此外，程序会在等量货币金额的输出位置旁边显示一面加拿大国旗。选择货币之前或之后，用户输入美元金额。一旦两个任务都完成，用户可点击“计算”按钮、“清除”按钮或“退出”按钮。点击“计算”按钮会将美元金额转换为所选货币的等量金额。点击“清除”按钮可重新设置货币选择、美元金额和等量货币金额及相关的标签。点击“退出”按钮，则结束该应用程序。这个例子很好地说明了UML描述和

一个面向对象的实现，第五部分还要讨论这个例子。

2.7 土星牌挡风玻璃雨刷

某些土星牌汽车的挡风玻璃雨刷是由带刻度盘的控制杆控制的。这种控制杆有四个位置：停止、间歇、低速和高速，刻度盘有三个位置，分别是数字1、2和3，刻度盘位置指示三种间歇速度，刻度盘的位置只有当控制杆在间歇位置上时才有意义，以下决策表给出了挡风玻璃雨刷对应控制杆和刻度盘的工作速度（每分钟摇摆次数）。

c1.控制杆	停止	间歇	间歇	间歇	低速	高速
c2.刻度盘	—	1	2	3	—	—
a1.雨刷	0	4	6	12	30	60

稍后在讨论交互测试时还将使用这个例子（请参阅第15章）。

2.8 参考文献

Brown, J.R. and Lipov, M., Testing for software reliability, *Proceedings of the International Symposium on Reliable Software,* Los Angeles, April 1975. pp. 518–527

Chellappa, Mallika, Nontraversible Paths in a Program, *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 6, June 1987, pp. 751–756.

Clarke, Lori A. and Richardson, Debra J., The application of error sensitive strategies to debugging, *ACM SIGSOFT Software Engineering Notes,* Vol. 8, No. 4, August 1983.

Clarke, Lori A. and Richardson, Debra J., A reply to Foster's comment on "The Application of Error Sensitive Strategies to Debugging," *ACM SIGSOFT Software Engineering Notes,* Vol. 9, No. 1, January 1984.

Gruenberger, F., Program testing, the historical perspective, in *Program Test Methods,* William C. Hetzel, Ed., Prentice-Hall, New York, 1973, pp. 11–14.

Hetzel, Bill, *The Complete Guide to Software Testing,* 2nd ed., QED Information Sciences, Inc., Wellesley, MA, 1988.

Inglis, Stuart J., *Planets, Stars, and Galaxies,* 4th Ed., John Wiley & Sons, New York, 1961.

Myers, Glenford J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.

Pressman, Roger S., *Software Engineering: A Practitioner's Approach,* McGraw-Hill, New York, 1982.

Topper, Andrew et al., *Structured Methods: Merging Models, Techniques, and CASE,* McGraw-Hill, New York, 1993.

2.9 练习

1. 请再次研究图2-1给出的传统三角形程序流图。变量match可以取值4或5吗？有可能

“执行”以下带编号方框的序列：1、2、5、6吗？

2. 第1章曾经讨论过程序规格说明和实现之间的关系。如果仔细地研究NextDate实现，就会看出一个问题。请看1个月有30天（4、6、9、11月）的CASE子句。没有日 = 31的特别行为。请讨论这种实现是否正确。请就2月对应的日 = 29取值处理的CASE子句进行类似的讨论。

3. 第1章提到测试用例的一部分是所预期的输出。NextDate的1812年6月31日测试用例的预期输出是什么？为什么？

4. 对三角形问题的一种常见补充是检查直角三角形。如果满足毕达哥拉斯（Pythagorean）关系：$c^2 = a^2 + b^2$，则三条边构成直角三角形。这种变化使要求按增序给出各条边变得很方便，即a <= b <= c。扩展Triangle3程序，以包含直角三角形特性。我们将在第二部分和第三部分的练习中使用这种扩展。

5. Triangle2程序怎样处理边长为-3、-3、5的情况？请采用第1章所给出的术语进行讨论。

6. 前一日函数是NextDate的逆函数。给定一个月份、日期、年，前一日会返回这一天的前一天的日期。请采用自己喜欢的语言开发一个前一日程序，把这作为一种扩展练习。

7. 部分GUI设计的艺术是防止用户输入错误。事件驱动的应用程序尤其易受输入错误的影响，因为事件可以以任何顺序发生。利用前面已经给出的伪代码定义，用户可以输入美元金额，然后点击计算按钮，而没有选择国家。类似地，用户可以选择一个国家并点击计算按钮，而没有输入美元金额。在面向对象的应用程序中，可以通过细心地实例化对象来控制这种情况。请修改GUI类的伪代码，以防止出现这两种情况。

8. CRC出版公司的Web站点（http://www.crcpress.com）提供了本书的一些补充软件。我在研究生软件测试课上使用了一系列练习，扩展练习的第一部分是使用naive.exe（在带Windows的PC机上运行）程序测试三角形、NextDate和佣金问题。Visual Basic程序可自解释。作为成为测试工艺师的一个起点，使用naive.exe以直觉（因此“自然”）方式测试这三个例子。每个程序都被插入缺陷。如果发现失效，请尽量推测其中基本的缺陷。请将所得到的结果与第5、6和9章的思想进行比较。

第3章 测试人员的离散数学

除了其他生命周期活动外，测试本身还要进行数学描述和分析。本章和下一章将为测试人员提供所需的数学知识。按照工艺师的说法，这里给出的数学方法就是工具，测试工艺师应该清楚地了解如何使用这些工具。通过这些工具，测试人员会变得严格、精确和高效，所有这些都会改进测试。本章题目中的“测试人员的”几个字很重要：本章是为只有一般数学基础，或已经忘了部分数学知识的测试人员编写的。严肃的数学家（或也许只是那些认真要求自己的数学家）可能会对这里的非正式讨论很不满意。如果读者已经了解本章的内容，可跳过本章直接研究图论。

一般来说，离散数学更适用于功能性测试，而图论更适合结构性测试。“离散”带来一个问题：什么是不离散的数学？数学上的反义词是连续，与微积分一样，这些开发人员（和测试人员）都很少能够用到。离散数学包括集合论、函数、关系、命题逻辑和概率论，以下分别讨论这些内容。

3.1 集合论

我们已经习惯了所有的严格和精确定义，但是令人尴尬的是，集合没有明确的定义。这确实很麻烦，因为集合论是关于数学的这两章内容的核心。对此，数学家进行了重要的区分：自然与不言自明的集合论。对于自然集合论，把集合看作是基本术语，就像几何学中的点和线的基本概念一样。有一些“集合”的同义词：聚集、组、束等，读者从中可以理解其含义。关于集合，重要的是它使我们能够作为一个单位，或一个整体引用多个事物。例如，我们可能要引用正好有30天的月份（在测试第2章的NextDate函数时，需要这个集合）。采用集合论表示法可以写为：

M1 = {4月，6月，9月，11月}

以上表示法读做“M1是元素为4月、6月、9月、11月的集合”。

3.1.1 集合成员关系

集合中的项叫做集合的元素或成员，这种关系采用符号 $\in$ 表示。这样我们可以有4月 $\in$ M1。如果事物不是集合成员，则使用符号 $\notin$ 表示，可以有12月 $\notin$ M1。

3.1.2 集合定义

集合有三种方式定义：简单列出集合的元素，给出辨别规则，或通过其他集合构建。列出元素方式适合只有少量元素的集合，或元素符合某种明显模式的集合。我们可以定义NextDate程序中的年份集合为：

Y = {1812，1813，1814，……，2011，2012}

通过列出元素定义集合时，与元素顺序没有关系，在讨论集合相等性时就会知道其中的原因。决策规则方法更复杂一些，这种复杂性既有优点又有代价。可以把NextDate的年份定义为：

Y = {年：1812≤年≤2012}

读做“Y是所有年的集合，使得（冒号读做“使得”）年份在1812（含）~2012（含）之间”。当使用决策规则定义集合时，该规则必须是无歧义的，给出年份的所有可能取值，因此可以判断某个年份是否在我们的集合Y中。

采用决策规则定义集合的优点是，无歧义要求使定义很清晰。有经验的测试人员都经历过“不可测试的需求”。很多时候出现这种情况的原因是，这种不能测试的需求源于歧义的决策规则。例如在我们的三角形程序中，假设定义一个集合：

N = {t：t是近似等边三角形}

我们可以说边长为（500，500，501）的三角形是N的一个元素，但是如何判断边长是（50，50，51）或（5，5，6）的三角形呢？

采用决策规则定义集合的第二个优点是，我们可能对元素很难列举的集合感兴趣。在佣金问题中，我们可能对以下集合感兴趣：

S = {销售：15%的佣金率适用于该销售额}

我们很难列出这个集合的元素，但是对于特定的销售额，可以很容易地利用这个决策规则。

决策规则的主要缺点是，有可能在逻辑上太复杂，尤其是当采用谓词演算量词∃（“存在”）和∀（“所有”）时。如果所有人都理解这种表示法，则精确性很有价值。但是客户往往被带有这些量词的语句搞糊涂。决策规则的第二个问题与自引用有关。自引用很有意思，但是测试人员很少使用。当决策规则引用自己的时候会出现循环问题。例如，Seville的理发师“是为所有不刮自己胡子的人刮胡子的人”。

3.1.3 空集

空集采用符号 ∅ 表示，在集合论中占有特殊位置。空集不包含元素。这时数学家会证明有关空集的大量事实：

- 空集是惟一的，即不会有两个空集（我们全盘接受地这种说法）
- ∅、{∅}、{{∅}}都是不同的集合（我们不需要这种事实）。

需要注意的是，如果集合被决策规则定义为永远失败，那么该集合就是空集。例如，

$$\varnothing = \{年：2012 \leqslant 年 \leqslant 1812\}$$

3.1.4 维恩图

当讨论已描述和已编程实现的行为时，常常像第1章一样画出维恩图。在维恩图中，集合被表示为一个圆圈，圆圈中的点表示集合元素。这样我们可以把有30天的月份的集合M1表示为图3-1。

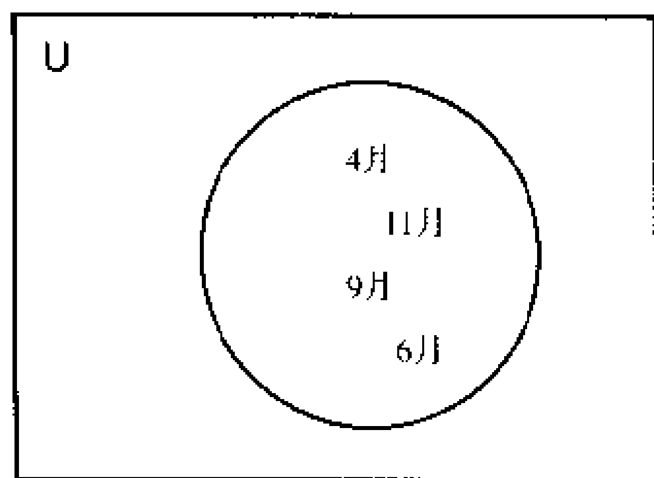


图3-1 有30天的月份集合的维恩图

维恩图以直观方式表示各种集合关系，但是也有一些小问题。怎么表示有限集合与无限集合？两种集合都可以采用维恩图表示，对于无限集合，不能假设所有内部点都对应一个集合元素。我们不必担心这一点，但是了解这种限制是有帮助的。有时我们发现为特定元素给出标签是有帮助的。

另一个小问题与空集有关。怎样显示集合或集合的一部分是空集？常见的方法是对空集部分加阴影，但是这又经常与阴影的其他应用矛盾，有时阴影是为了突出所感兴趣的部分。最好的办法是给出图例，表明阴影部分的实际含义。

常常需要所讨论的所有集合看做是某个更大集合的子集，这个更大集合叫做论域空间。第1章在选择所有程序行为为论域空间时已经这样做了。论域空间通常可以通过所给出的集合猜出。在图3-1中，大多数人都会把论域空间看做是一年中所有月份的集合。测试人员应该认识到假设的论域空间常常会造成混乱，因此论域空间常常是在客户和开发人员之间引起误解的一个潜在原因。

3.1.5 集合操作

集合论的表示能力主要体现在集合基本操作上：并、交和补。人们还使用其他便利的操作：相对补、对称差和笛卡儿积。以下定义这些操作。在这些定义中，我们首先给出两个

集合，即某个论域空间U所包含的A和B。这些定义使用来自谓词演算的逻辑连接符、与（∧）、或（∨）、异或（⊕）和非（¬）。

定义

给定集合A和B，

其并是集合$A \cup B = \{x: x \in A \vee x \in B\}$。

其交是集合$A \cap B = \{x: x \in A \wedge x \in B\}$。

A的补是集合$A' = \{x: x \notin A\}$。

B针对A的相对补是集合$A - B = \{x: x \in A \wedge x \notin B\}$。

A和B的对称差是集合$A \oplus B = \{x: x \in A \oplus x \in B\}$。

这些集合的维恩图如图3-2所示。

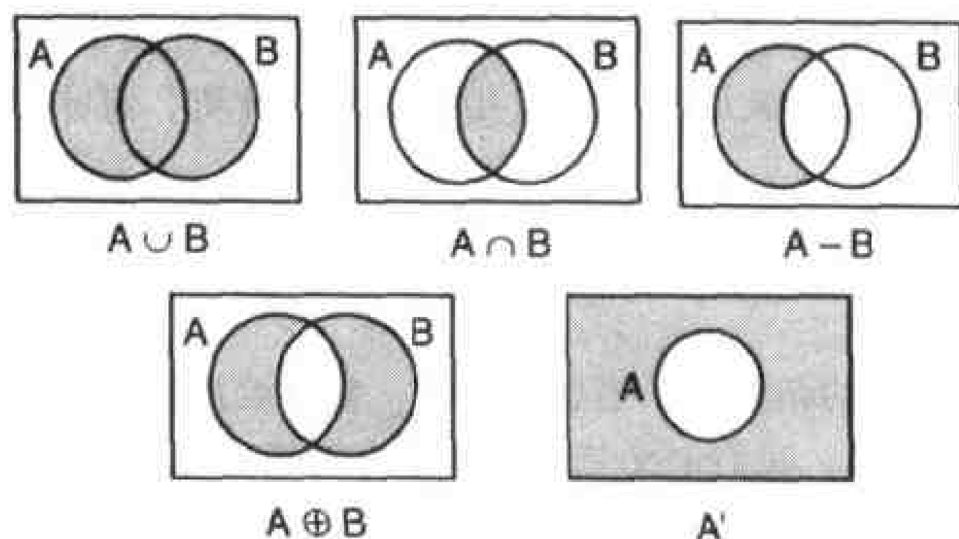


图3-2 基本集合的维恩图

维恩图的直观表示能力，对于描述测试用例之间和被测软件之间的关系非常有用。通过观察图3-2中的维恩图可以猜出：

$$A \oplus B = (A \cup B) - (A \cap B)$$

这个结论没错，采用谓词逻辑可以证明。

维恩图还在软件开发中的其他地方使用：结合有向图，维恩图可以构成状态图表示法的基础，这是由CASE技术支持的最严格的规格说明技术之一。状态图也是Rational公司和对象管理集团统一建模语言UML所选的控制表示法。

两个集合的笛卡儿积（又叫做叉积）更加复杂，它取决于有序对偶的概念，即两个元素集合中的元素顺序是重要的。无序和有序对偶的表示法一般是：

无序对偶：(a, b)

有序对偶：<a, b>

两者的差别是，对于$a \neq b$，

(a, b) = (b, a)，但是

$\langle a, b\rangle \neq \langle b, a\rangle$

这种差别对于第4章的内容很重要，正如我们将会看到的，一般和直接图之间的基本差别恰恰是无序和有序对偶之间的差别。

定义

两个集合A和B的笛卡儿积，是集合

$$A \times B = \{<x, y>: x \in A \wedge x \in B\}$$

维恩图不能显示笛卡儿积，因此举一个简单的例子。集合A = {1, 2, 3}和B = {w, x, y, z}的笛卡儿积是集合：

$$A \times B = \{<1, w>, <1, x>, <1, y>, <1, z>, <2, w>, <2, x>, <2, y>, <2, z>, <3, w>, <3, x>, <3, y>, <3, z>\}$$

笛卡儿积与运算有直观的联系。集合A的势是A中的元素数，采用|A|表示。（有些人喜欢采用Card（A）表示。）对于集合A和B，|A × B| = |A| × |B|。当我们在第5章研究功能性测试时，将使用笛卡儿积描述具有多个输入变量的程序的测试用例。笛卡儿积的乘法性质，意味着这种形式的测试会生成大量测试用例。

3.1.6 集合关系

我们使用集合操作通过现有集合构建有意思的新集合。当这样做时，常常希望知道新集合和老集合之间的关联方式。给定两个集合A和B，我们定义三种基本集合关系：

定义

A是B的子集，记做$A \subseteq B$，当且仅当$a \in A \Rightarrow a \in B$。

A是B的真子集，记做$A \subset B$，当且仅当$A \subseteq B \wedge B - A \neq \varnothing$。

A和B是相等集合，记做A = B，当且仅当$A \subseteq B \wedge B \subseteq A$。

用普通语言描述就是，如果A的每个元素也是B的元素，集合A是集合B的子集。为了成为B的真子集，A必须是B的子集，而且B中必须存在不是A元素的元素。最后，如果集合A和B互为子集，则A和B相等。

3.1.7 子集划分

集合的一个划分是一种非常特殊的情况，对于测试人员非常重要。划分在很多方面可以和日常生活类比：通过划分将一间办公室分成独立的个人办公室；根据政策划分，一个州可以分成司法区。在这两种划分中，请注意“划分”的含义是将一个整体分成小块，使得所有事物都在某个小块中，不会遗漏。更形式化的描述是：

定义

给定集合B，以及B的一组子集A1、A2、……、An，这些子集是B的一个划分，当且仅当：

$A1 \cup A2 \cup \cdots \cup An = B$，且

$i \neq j \Rightarrow Ai \cap Aj = \varnothing$

由于一个划分是一组子集，因此我们常常把单个子集看做是划分的元素

这个定义的两部分对于测试人员很重要，第一部分保证B的所有元素都在某个子集中，第二部分保证B没有元素在两个子集中。这种情况与司法区例子有很好的对应关系：每个人都由某个立法委员代表，并且没有人由两个立法委员代表。拼图游戏是划分的另一个很好的例子，事实上，划分的维恩图常常类似拼图，如图3-3所示。

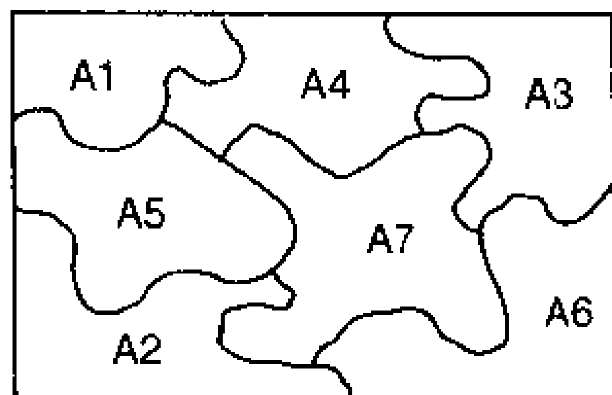


图3-3 划分的维恩图

划分对测试人员很有用，因为两个界定性质会产生重要保证：完备性（任何事物都在某处）和无冗余性。当研究功能性测试时，我们会看到功能性测试的固有弱点是漏洞和冗余性：有些内容没有被测试，而另外一些内容被测试多次。功能性测试的主要困难之一，就是找出合适的划分。例如，在三角形程序中，论域空间是所有正整数的三元组。（请注意，这实际上是正整数集合自身的三次笛卡儿积。）我们有三种方法可以划分这个论域空间：

1. 三角形和非三角形。
2. 等边三角形、等腰三角形、不等边三角形和非三角形
3. 等边三角形、等腰三角形、不等边三角形、直角三角形和非三角形。

初看起来这些都划分没有问题，但是仔细研究会发现最后一个划分有一个问题，即不等边三角形和直角三角形并不是不相交的（边长为3、4、5的三角形是不等边的直角三角形。）

3.1.8 集合恒等式

集合操作和关系合在一起，会产生一种重要的集合恒等式类，可以用于代数级地简化复杂集合的表示。数学系学生通常必须导出所有这些恒等式，这里只将其列出，并（偶尔）使用。

名 称	表 达 式
等同律	$A \cup \emptyset = A$
	$A \cap U = A$
支配律	$A \cup U = U$
	$A \cap \emptyset = \emptyset$

（续）

名称	表达式
幂等律	$A \cup A = A$
	$A \cap A = A$
求反律	$(A')' = A$
交换律	$A \cup B = B \cup A$
	$A \cap B = B \cap A$
结合律	$A \cup (B \cup C) = (A \cup B) \cup C$
	$A \cap (B \cap C) = (A \cap B) \cap C$
分配律	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
迪摩根定律	$(A \cup B)' = A' \cap B'$
	$(A \cap B)' = A' \cup B'$

3.2 函数

函数是软件开发和测试的核心概念。例如，整体功能分解范例就隐含地使用了函数的数学概念。我们在这里明确地讨论这种概念，因为所有功能性测试的基础都是函数。

非正式地说，函数与集合元素关联。例如，在NextDate程序中，给定日期的函数是NextDate日期；在三角形问题中，三个输入整数的函数是以这些长度为边的三角形种类；在佣金问题中，销售商的佣金是销售额的函数，销售额又是所售出枪机、枪托和枪管数量的函数。ATM系统中的函数要复杂得多，毫不奇怪，这会增加测试的复杂性。

任何程序都可以看做是将其输出与其输入关联起来的函数。用数学公式表示函数，输入是函数的定义域，输出是函数的值域。

定义

给定集合A和B，函数f是A × B的一个子集，使得对于a_i、$a_j \in A$，b_i、$b_j \in B$，$f(a_i) = b_i$，$f(a_j) = b_j$，$b_i \neq b_j \Rightarrow a_i \neq a_j$。

像这样的形式化定义不够简洁，因此需要进一步研究。函数f的输入是集合A的元素，f的输出是B的元素。这个定义是说，函数f“表现良好”是指A中的元素永远也不与B的多个元素关联。（如果出现这种情况，那么应该怎样测试这类函数呢？这是非确定性的一个例子。）

3.2.1 定义域与值域

在刚刚给出的定义中，集合A是函数f的定义域，集合B是值域。由于输入和输出具有某种“自然”顺序，因此很容易进一步说函数f是一个有序对偶的集合，其中第一个元素来自定义域，第二个元素来自值域。以下是函数的两种常见表示法：

$f: A \to B$

$f \subseteq A \times B$

在这个定义中没有对集合A和B进行任何限制。我们可以有A = B，A或B可以是其他集合的笛卡儿积。

3.2.2 函数类型

函数进一步通过映射细节描述。在以下定义中，首先给出函数f：A → B，并且定义集合：

$$f(A) = \{b_i \in B: b_i = f(a_i) \text{ 对于某个 } a_i \in A\}$$

这个集合有时记做A在f下的映象。

定义

f是从A到B的上函数，当且仅当f（A）= B

f是从A到B的中函数，当且仅当f（A）⊂ B。（请注意这里的真子集！）

f是从A到B的一对一函数，当且仅当对于所有a_i、$a_j \in A$，$a_i \neq a_j \Rightarrow f(a_i) \neq f(a_j)$

f是从A到B的多对一函数，当且仅当存在a_i、$a_j \in A$，$a_i \neq a_j$使得$f(a_i) = f(a_j)$

再用普通语言解释一下，如果f是从A到B的上函数，则B的每个元素都与A的某个元素关联。如果f是A到B的中函数，则B中至少有一个元素与A的某个元素关联。一对一函数保证某种形式的惟一性：不同定义域元素永远不会映射到相同的值域元素上。（请注意，这是前面描述过的“良好行为”属性的逆。）如果函数不是一对一的，则是多对一的，即多个定义域元素可以映射到相同值域元素上。采用这些术语说，“良好行为”需求可防止函数出现一对多情况。熟悉关系数据库的测试人员会意识到这些可能性（一对一、一对多、多对一和多对多）都涉及关系。

再回到我们的测试例子上，假设取A、B和C作为NextDate程序的日期集合，其中：

A = {日期：1812年1月1日 ≤ 日期 ≤ 2012年12月31日}

B = {日期：1812年1月2日 ≤ 日期 ≤ 2013年1月1日}

C = A ∪ B

现在，NextDate：A → B是一个一对一的上函数，NextDate：A → C是一个一对一的中函数。

对于NextDate来说，多对一没有意义，但是很容易看出三角形问题怎样成为多对一。如果函数是一对一的上函数，例如前面讨论过的NextDate：A → B，那么定义域中的每个元素都恰好对应值域中的一个元素。反之，值域中的每个元素也恰好对应定义域中的一个元素。当出现这种情况时，总是可以找到一个从值域回到定义域的一对一的逆函数（请参阅第2章中的前一日问题）。

所有这些对于测试都很重要。中函数与上函数，意味着基于定义域还是基于值域的功能

性测试，一对一函数要求比多对一函数要多得多的测试。

3.2.3 函数合成

假设我们有集合和函数，使得一个函数的值域是另一个函数的定义域：

$$f: A\rightarrow B$$
$$g: B\rightarrow C$$
$$h: C\rightarrow D$$

如果出现这种情况，则可以合成函数。为此，设引用集合定义域和值域的特定元素$a\in A$、$b\in B$、$c\in C$、$d\in D$，并假设$f(a)=b$、$g(b)=c$和$h(c)=d$，则函数g和f的合成为：

$$\begin{aligned} h\circ g\circ f(a) &= h(g(f(a))) \\ &= h(g(b)) \\ &= h(c) \\ &= d \end{aligned}$$

函数合成在软件开发中是非常常见的实践，在定义过程和子过程的过程中是固有的。佣金例子中就有一个这样的例子：

$$f_1(\text{枪机，枪托，枪管})=\text{销售额}$$
$$f_2(\text{销售额})=\text{佣金}$$

函数的合成链对于测试人员可能是个问题，尤其是当一个函数的值域是函数合成链"下一个"函数定义域的真子集。图3-4通过数据流图显示了在程序中怎样出现这种情况。

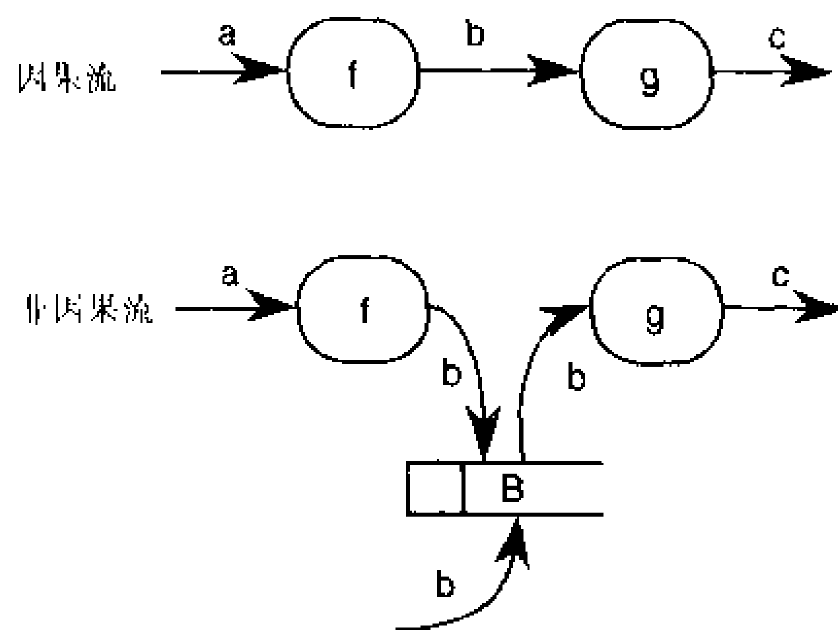


图3-4 数据流图中的因果流和非因果流

在因果流中，合成$g\cdot f(a)$（我们知道它是产生c的$g(b)$）是与装配线很相似的过程。在非因果流中，对于数据库B的多个b取值来源的可能性会为测试人员带来两个问题。B取值

的多个来源会产生定义域/值域兼容性问题；即使不会出现这种问题，也可能出现与b取值有关的时序异常问题。（如果g使用了一个“旧”b值会出现什么情况？）

可以使用合成的一个特例，以便以一种奇特的方式帮助测试人员。我们曾经讨论过一对一的上函数怎样永远有一个逆函数的问题。这种逆函数是惟一的，并保证存在（数学家也同样会证明这个问题）。如果f是从A到B上的一对一函数，则这个惟一逆函数记做f^{-1}。对于$a \in A$和$b \in B$，$f^{-1} \cdot f(a) = a$和$f \cdot f^{-1}(b) = b$。NextDate和前一日函数就是这种逆函数。对测试人员有帮助的方面是，对于给定函数，其逆函数充当某种“交叉检查”，而这常常可以加快功能性测试用例的标识。

3.3 关系

函数是关系的一种特例：两者都是某个笛卡儿积的子集，但是对于函数，我们有“良好行为”需求，即定义域元素不能与多个值域元素关联。日常使用模式可以证明这一点：当我们说某事物是其他事物的“函数”，意思是说存在确定的关系表示。并不是所有关系都严格地是函数。考虑病人集合和内科医生集合之间的映射。一个病人可能由多个内科医生治疗，一个内科医生可能治疗多个病人，这是一种多对多映射。

3.3.1 集合之间的关系

定义

给定两个集合A和B，关系R是笛卡儿积$A \times B$的一个子集。

有两种表示法很常见，如果希望描述整个关系，则通常只写$R \subseteq A \times B$。对于特定元素$a_i \in A$、$b_i \in B$，我们记做$a_i R b_i$。有关关系的论述这里都省略了，我们对关系感兴趣，是因为关系是数据建模和面向对象分析的基础。

接下来，我们必须解释一个被大量使用的术语——势。前面已经讨论过，势在用于集合时，是指集合中的元素个数。由于关系也是集合，因此可以期望关系的势是指有多少有序对偶在集合$R \subseteq A \times B$中。但是，情况并不是这样。

定义

给定两个集合A和B，一个关系$R \subseteq A \times B$，关系R的势是：

一对一势，当且仅当R是A到B的一对一函数。

多对一势，当且仅当R是A到B的多对一函数。

一对多势，当且仅当至少有一个元素$a \in A$在R中的两个有序对偶中，即$(a, b_i) \in R$和$(a, b_j) \in R$。

多对多势，当且仅当至少有一个元素$a \in A$在R中的两个有序对偶中，即$(a, b_i) \in R$和$(a, b_j) \in R$，并且至少有一个元素$b \in B$在R中的两个有序对偶中，$(a_i, b) \in R$和$(a_j, b) \in R$。

函数映射到值域上或值域中之间的差别可以与关系类比，这就是参与概念。

定义

给定两个集合A和B，一个关系R $\subseteq$ A × B，关系R的参与是：

全参与，当且仅当A中的所有元素都在R的某个有序对偶中；
部分参与，当且仅当A中有元素不在R的有序对偶中，
上参与，当且仅当B中的所有元素都在R的某个有序对偶中，
中参与，当且仅当B中有元素不在R的有序对偶中。

采用一般语言说就是，关系是全参与，如果它适用于A的每个元素；关系是部分参与，如果它不适用于A的所有元素。描述这种差别的另一种方式是强制参与和可选参与。类似地，关系是上参与，如果它适用于B的每个元素；关系是中参与，如果它不适用于B的所有元素。全参与/部分参与和上参与/中参与的平行性很奇怪，值得在这里特别讨论一下。从关系数据库理论的立场看，出现这种平行性没有什么原因。事实上，有引人注目的原因避免出现这种差别。数据建模本质上是陈述性的，而过程建模本质上是强制性的。术语的平行集合要求在关系上有方向，但是事实上不需要这种方向性。部分原因可能是因为笛卡儿积由有序对偶组成，明显地拥有第一和第二元素。

到目前为止，我们只考虑了两个集合之间的关系。将关系扩展到三个或更多集合，要比纯粹的笛卡儿积复杂。例如，假设有三个集合A、B和C，以及一个关系R $\subseteq$ A × B × C。我们希望关系严格地定义在三个元素上，还是定义在一个元素和一个有序对偶上（这里有三种可能）？沿着这种思路，还会涉及到势和参与的定义。对于参与来说是直接而简明的，但是势是二元性质的。（例如，假设关系是从A到B的一对一关系，并且是A到C的多对一关系。）我们在第1章讨论已描述、已实现和已测试程序行为时，曾经讨论过一种三向关系。我们希望在测试用例和规格说明–实现对偶之间有某种形式的全参与，在讨论功能性和结构性测试时还要讨论这个问题。

测试人员需要关心关系的定义，因为关系的定义直接与被测软件性质有关。例如，上参与/中参与的差别，直接与我们称之为基于输出的功能性测试有关。强制–可选差别是例外处理的基础，对于测试人员也很有用。

3.3.2 单个集合上的关系

我们使用两种重要的数学关系，即排序关系和等价关系。这两种数学关系都定义在单个集合上，都使用了关系的具体性质。

设A是一个集合，设R $\subseteq$ A × A是定义在A上的一个关系，<a, a>、<a, b>、<b, a>、<b, c>、<a, c> $\in$ R。关系具有四个特殊属性：

定义

关系R $\subseteq$ A × A是：

自反的，当且仅当所有$a \in A$，$<a, a> \in R$，

对称的，当且仅当$<a, b> \in R \Rightarrow <b, a> \in R$

反对称的，当且仅当$<a, b>$、$<b, a> \in R \Rightarrow a = b$

传递的，当且仅当$<a, b>$、$<b, c> \in R \Rightarrow <a, c> \in R$

家庭关系是这些特性的很好例子。读者可以考虑以下关系，并自己确定关系所具有的属性：兄弟关系、同胞关系、祖先关系。下面定义两个重要关系。

定义

关系$R \subseteq A \times A$是排序关系，如果R是自反、反对称和传递的。

排序关系有方向。一些常见的排序关系有Older than（较老）、$\geqslant$、$\Rightarrow$和Ancestor of（祖先）（自反性通常要求说一些怪话——我们实际上应该说“不比……年轻”和“不是……的后代”）。排序关系在软件中很常见：数据访问手段、杂凑代码、树型结构和数组，都利用了排序关系。

给定集合的幂集合，是给定集合的所有子集的集合。集合A的幂集合记做P（A）。子集关系$\subseteq$是P（A）上的一种排序关系，因为它是自反的（任何集合都是其自身的一个子集），是反对称的（集合相等的定义），并且是传递的。

定义

关系$R \subseteq A \times A$是等价关系，如果R是自反、对称和传递的。

数学中有大量等价关系：相等和重叠就是两个立即可以想到的例子。假设有集合B上的某个划分A_1、A_2、…、A_n，我们说B的两个元素b_1和b_2是相关的（即$b_1 R b_2$），如果b_1和b_2是在相同的划分元素中。这个关系是自反的（任何元素都在其自己的划分中），是对称的（如果b_1和b_2是在某个划分元素中，那么b_2和b_1也在这个划分元素中），是传递的（b_1和b_2是在同一个集合中，而且如果b_2和b_3也在同一个集合中，则b_1和b_3在同一个集合中）。通过划分定义的关系叫做由划分归纳的等价关系。逆过程也同样存在。如果从定义在一个集合上的等价关系开始，则可以根据与该等价关系相关的元素定义子集。这就是划分，叫做由等价关系归纳的划分。这种划分中的集合叫做等价类。最终结果是划分和等价关系可以相互交换，而这一点对于测试人员来说是很重要的概念。前面已经介绍过，划分有两个性质，即完备性和无冗余性。将这些性质带到测试领域中，可使测试人员做出关于软件已经测试的广度的有力、绝对的说明。不仅如此，只测试等价类中的一个元素，并假设剩余的元素有类似的测试结果，可大大提高测试效率。

3.4 命题逻辑

我们已经在使用命题逻辑表示法了。如果读者对以前的这种使用定义不太理解，这没有什么奇怪的。集合论和命题逻辑具有一种鸡和蛋的关系——很难确定应该先讨论哪一个。就

像集合是基本术语，因此不能定义一样，命题也是基本术语。命题是要么真要么假的句子，我们叫做命题的真值。不仅如此，命题是无歧义的：给定一个命题，总是能够确定它是真还是假。句子“数学很难”就不是一个命题，因为有歧义。我们通常采用小写字母p、q和r表示命题。命题逻辑具有与集合论非常类似的操作、表达式和标识（实际上两种是同构的）

3.4.1 逻辑操作符

逻辑操作符（又叫做逻辑连接符或操作）根据它们对命题真值的作用来定义。也就是说，只使用两个值：T（代表真）和F（代表假）。也可以以同样方式定义算术操作符（实际上这就是孩子们考虑算术的方式），但是表格会变得太大。三种基本逻辑操作符是与（∧）、或（∨）和非（¬）。这些操作符有时又叫做合取、析取和非。非是惟一一个一元（一个操作数）逻辑操作符，其他都是二元操作符。

p	q	$p \wedge q$	$p \vee q$	$\neg p$
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

合取和析取在日常生活中很常见：只有当所有组件都为真时，合取才为真；至少有一个组件为真时，析取就为真。非也与我们的预期一样。人们经常使用两种连接：异或（⊕）和如果-则（IF-THEN）（→）。定义如下：

p	q	$p \oplus q$	$p \to q$
T	T	F	T
T	F	T	F
F	T	T	T
F	F	F	T

只有当一个命题为真时，异或为真，而析取（也就是或）当两个命题都为真时也为真。IF-THEN连接常常会带来困难。最简单的方法是只把它看做是一种定义。但是由于其他连接都可以很好地与自然语言对应，因此对IF-THEN也会有类似的预期。关于IF-THEN的简单回答与演绎过程密切相关：在有效的演绎推论中，我们可以说“如果有前提，则会得到结论”，而蕴涵语句将是重言式。

3.4.2 逻辑表达式

采用逻辑操作符构建逻辑表达式的方式，与使用算术操作符构建代数表达式的方式完全一样。我们可以定义有括号时操作符的一般使用顺序，也可以使用优先级顺序（非最先，然后是合取，最后是析取）。给定一个逻辑表达式，总是能够通过由括号确定的顺序“构造”出真值表来。例如，表达式$\neg((p \to q) \wedge (q \to p))$具有以下真值表：

p	q	$p \to q$	$q \to p$	$(p \to q) \wedge ((q \to p)$	$\neg((p \to q) \wedge ((q \to p))$
T	T	T	T	T	F
T	F	F	T	F	T
F	T	T	F	F	T
F	F	T	T	T	F

3.4.3 逻辑等价

算术相等和相同集合的概念，可与命题逻辑类比。请注意，表达式$\neg((p \to q) \wedge (q \to p))$和$p \oplus q$具有相同的真值表。这意味着不管基本命题p和q取什么真值，这些表达式都永远具有相同的真值。这种性质可以用多种方式定义，我们选用最简单的定义

定义

两个命题p和q是等价的（记做$p \Leftrightarrow q$），当且仅当其真值表相同。

顺便提一下，作为“如果且仅仅如果”的缩写“当且仅当”有时记做双向条件，因此，命题p当且仅当q实际上是$(p \to q) \wedge (q \to p)$，记做$p \leftrightarrow q$。

定义

永远为真的命题是重言式，永远为假的命题是矛盾式。

为了成为重言式或矛盾式，命题必须包含至少一个连接词和两个或多个基本命题。我们有时把重言式记做命题T，把矛盾式记做命题F。下面给出一些直接与集合类比的定律

定律	表达式
等同律	$p \wedge T \Leftrightarrow p$
	$p \vee F \Leftrightarrow p$
支配律	$p \vee T \Leftrightarrow T$
	$p \wedge F \Leftrightarrow F$
幂等律	$p \wedge p \Leftrightarrow p$
	$p \vee p \Leftrightarrow p$
求反律	$\neg(\neg p) \Leftrightarrow p$
交换律	$p \wedge q \Leftrightarrow q \wedge p$
	$p \vee q \Leftrightarrow q \vee p$
结合律	$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$
	$p \vee (q \vee r) \Leftrightarrow (p \vee q) \vee r$
分配律	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$
	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
迪摩根定律	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$

3.5　概率论

在研究软件测试时，有两种情况需要使用概率论，一是研究语句执行特定路径的概率，二是将其泛化为叫做操作剖面的流行行业概念（请参阅第14章）。由于概率论在本书中的使用很有限，因此这里只介绍一些初步知识。

与集合论和命题逻辑一样，我们首先从基本概念开始，即事件的概率。以下是经典教科书中的定义（Rosen，1991）：

结果可能性相等的有限样本空间S中的事件E的概率，是p（E）=|E|/|S|。

这个定义依赖输出结果的经验，样本空间是所有可能结果的集合，事件是结果的子集。这个定义存在循环：什么是“可能性相等的”结果？我们设定这些结果具有相等的概率，这样概率就要由自身定义。

法国数学家拉普拉斯在两个世纪之前就致力于概率的合理定义。为了解释这个定义，事件发生的概率是期望事件发生方式的数量，除以总的可能方式数量（期望的方式和不期望的方式）。当我们从袋子中掏弹球时，这个定义很适用（概率论学者非常关注自己的弹球，可能其中曾有过教训），但是不能推广到很难枚举各种可能性的情况。

我们将利用集合论和命题逻辑的（经过改进的）能力，得到更一致的定义。作为测试人员，我们关心发生了的事情，我们把这些事情叫做事件，并说所有事件的集合是我们的论域空间。接下来，我们用命题定义事件，使得命题能够引用论域空间中的元素。现在，对于某个论域空间U和某个关于U的元素的命题p，我们有以下定义：

定义

命题p的真值集合T记做T（p），是p为真的论域空间U中的所有元素的集合。

命题要么真要么假，因此命题p将论域空间划分为两个子集，即T（p）和（T（p））'，其中，T（p）∪（T（p））'=U。请注意，（T（p））'与T（¬p）相同。真值集合有利于在集合论、命题逻辑和概率论之间清晰地映射。

定义

命题p为真的概率记做Pr（p），是|T（p）|/|U|。

通过这个定义，拉普拉斯的“期望方式的个数”变成真值集合T（p）的势，方式总数变成论域空间的势。这又产生一个连接：由于重言式的真值集合是论域空间，矛盾式的真值集合是空集，因此∅和U的概率分别是0和1。

NextDate问题是一个很好的例子。考虑变量月份和命题：

p（m）：m是一个有30天的月份

论域空间是集合U={1月，2月，……，12月}，p（m）的真值集合是集合：

T（p（m））={4月，6月，9月，11月}

这样，给定月份有30天的概率是：

$$Pr(p(m)) = |T(p(m))|/|U| = 4/12$$

论域空间角色有一个微妙的地方，这是在测试中使用概率论的工艺的一部分——只要选择正确的论域空间。假设我们想知道月份是2月的概率。快速回答是1/12。现在假设要计算有恰好有29天的月份的概率。这种计算不那么容易——需要一个既包括闰年也包括非闰年的论域空间。我们可以使用重叠算法，并选择一个包含4个连续年的论域空间，例如1991、1992、1993和1994。这个论域空间包含48个“月份”。在这个论域空间中，包含29天的月份的概率是1/48。另一种可能是使用NextDate程序两个世纪的值域，其中的2000年不是闰年。这种计算会稍微降低一点有29天的月份的概率。结论：选取正确的论域空间很重要。更大的结论：避免“转变论域空间”甚至更重要。

以下是我们将不加证明地使用的有关概率的一些事实。这些事实涉及给定论域空间、命题p和q，真值集合是T(p)和T(q)：

$$Pr(\neg p) = 1 - Pr(p)$$

$$Pr(p \wedge q) = Pr(p) \times Pr(q)$$

$$Pr(p \vee q) = Pr(p) + Pr(q) - Pr(p \wedge q)$$

这些事实，结合集合论和命题恒等式表，为操作概率表达式提供了强有力的代数能力。

3.6 参考文献

Rosen, Kenneth H., *Discrete Mathematics and Its Applications*, McGraw-Hill, New York, 1991.

3.7 练习

1. 集合操作和命题逻辑中的逻辑连接符具有很深的联系（同构）。

操作	命题逻辑	集合论
合取	或	并
析取	与	交
非	非	补
蕴涵	如果，那么	子集
	异或	对称差

a. 用语言表示$A \oplus B$

b. 用语言表示$(A \cup B) - (A \cap B)$

c. 证明$A + B$和$(A \cup B) - (A \cap B)$是同一个集合。

d. $A \oplus B = (A - B) \cup (B - A)$成立吗？

e. 在上面的表格中，应该为空格中的内容起什么名字呢？

2. 在美国的很多地方，要根据不同对象收取房地产税，例如学校地区、防火地区、城镇等。请讨论这些征税对象是否构成一个州的划分。50个州是否构成美国的一个划分？（哥伦比亚地区如何处理？）

3. 兄弟关系是在所有人的集合上的等价关系吗？同胞关系呢？

第4章
测试人员的图论

图论是拓扑学的一个分支，有时叫做“橡胶片几何学”。这很有趣，因为拓扑学的橡胶片部分几乎与图论无关。不仅如此，图论中的图不涉及读者可能预期的轴、刻度、点和曲线。不管其来源是什么，对于计算机科学来说，图论都可能是最有用的数学中部分，要比微积分重要得多，但是图论却没有普遍讲授。我们对图论的讨论，将遵循“纯数学”精神：定义尽可能不给出具体解释。推迟解释会使以后的解释更具灵活性，就像良好定义的抽象数据类型很有益于重用一样。

本节将使用两种基本图：无向图和有向图。由于后者是前者的特例，因此我们首先介绍无向图，这样在讨论有向图时，可以继承很多概念。

4.1 图

图（又叫做线性图）是一种由两个集合定义的抽象数学结构，即一个节点集合和一个构成节点之间连接的边集合。计算机网络是一个很好的图的例子。图的更形式化的定义如下：

定义

图G＝（V，E）由节点的有限（并且非空）集合V和节点无序对偶集合E组成，

$$V = \{n_1, n_2, \ldots, n_m\}$$

和

$$E = \{e_1, e_2, \ldots, e_p\}$$

其中每条边 $e_k = \{n_i, n_j\}$，n_i、$n_j \in V$。第3章曾经介绍过，集合$\{n_i, n_j\}$是一个无序对偶，有时记做（n_i，n_j）。

节点有时又叫做顶点，边有时又叫做弧，我们有时把节点叫做弧的端点。图的常见可视形式用圆圈表示节点，用节点对之间的连线表示边，如图4-1所示。我们将多次使用这张图来举例，因此请读者花几分钟熟悉一下图4-1。

如图4-1所示，节点和边集合为：

$$V = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$$
$$E = \{e_1, e_2, e_3, e_4, e_5\}$$
$$= \{(n_1, n_2), (n_1, n_4), (n_3, n_4), (n_2, n_5), (n_4, n_6)\}$$

为了定义特定的图，必须首先定义一组节点，然后定义节点对偶之间的一组边。我们通常把节点看做是程序语句，并且有各种边，例如表示控制流或定义/使用关系。

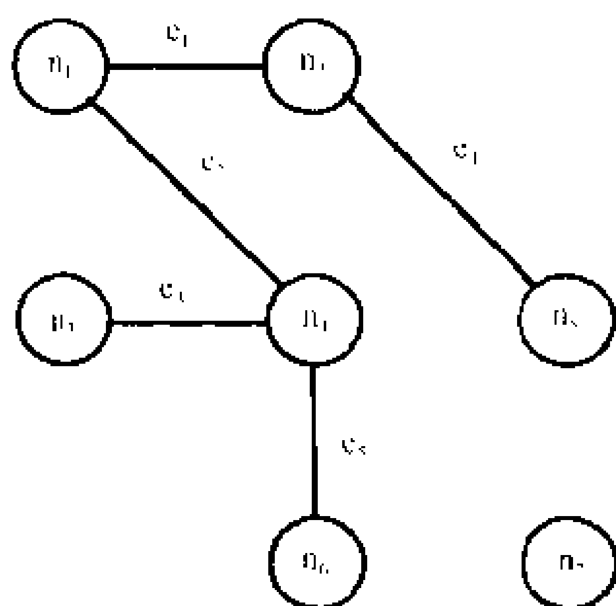


图4-1 有7个节点和5条边的图

4.1.1 节点的度

定义

图中节点的度是以该节点作为端点的边的条数。我们把节点n的度记做deg（n）。

我们可以说，节点的度表示它在图中的"流行程度"。事实上，社会学家使用图描述社会交互作用，其中的节点是人，边常常指诸如"友谊"、"与……通信"等关系。如果图中的节点表示对象，边表示消息，则节点（对象）的度表示适合该对象的集成测试范围。

图4-1中的节点的度是：

$$\deg(n_1)=2$$
$$\deg(n_2)=2$$
$$\deg(n_3)=1$$
$$\deg(n_4)=3$$
$$\deg(n_5)=1$$
$$\deg(n_6)=1$$
$$\deg(n_7)=0$$

4.1.2 关联矩阵

图不一定非要通过图形表示，图通过关联矩阵可以充分表示。这种概念对于测试人员来说正逐渐变得非常重要，因此这里要将这种概念形式化。当要对图进行具体解释时，关联矩阵永远为新的解释提供有用信息。

定义

拥有m个节点和n条边的图G=（V，E）的关联矩阵是一种m×n矩阵，其中第i行第j列的元素是1，当且仅当节点i是边j的一个端点，否则该元素是0。

图4-1中图的关联矩阵是：

	e_1	e_2	e_3	e_4	e_5
n_1	1	1	0	0	0
n_2	1	0	0	1	0
n_3	0	0	1	0	0
n_4	0	1	1	0	1
n_5	0	0	0	1	0
n_6	0	0	0	0	1
n_7	0	0	0	0	0

下面可以通过观察关联矩阵对图进行一些研究。首先，请注意任何列的表项和为2。这是因为每条边都恰好有两个端点。如果关联矩阵的列的和不是2，则一定出现了错误。因此，计算列的和，是一种完整性检查，类似对偶检查。第二，可以看到行的和是节点的度。如果节点的度是0，例如节点n_7，则说这个节点是孤立的。（这可以对应不可到达的节点，或已经包含但从来不使用的对象。）

4.1.3 相邻矩阵

图的相邻矩阵是对关联矩阵的很有用的补充。由于相邻矩阵考虑的是连接，因此相邻矩阵是很多较新的图论概念的基础。

定义

拥有m个节点和n条边的图G＝（V，E）的相邻矩阵是一种m×m矩阵，其中第i行第j列的元素是1，当且仅当节点i和节点j之间存在一条边，否则该元素是0。

相邻矩阵是对称的（元素i，j永远等于元素j，i），行的和是节点的度（与关联矩阵一样）。

图4-1中图的相邻矩阵是：

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	0	1	0	1	0	0	0
n_2	1	0	0	0	1	0	0
n_3	0	0	0	1	0	0	0
n_4	1	0	1	0	0	1	0
n_5	0	1	0	0	0	0	0
n_6	0	0	0	1	0	0	0
n_7	0	0	0	0	0	0	0

4.1.4 路径

我们先研究一下如何使用图论。测试的结构化方法（请参阅第三部分）都以程序中的路径类型为核心。这里我们定义（不加解释）图中的路径。

定义

路径是一系列的边，对于序列中的任何相邻边对偶e_i、e_j，边都拥有相同的（节点）端点。

路径可以描述为一系列边，也可以描述为一系列节点。一般更常见的是节点序列。

图4-1中图的一些路径是：

路　径	节点序列	边序列
n_1和n_5之间	n_1，n_2，n_5	e_1，e_4
n_6和n_5之间	n_6，n_4，n_1，n_2，n_5	e_5，e_2，e_1，e_4
n_3和n_2之间	n_3，n_4，n_1，n_2	e_3，e_2，e_1

路径可以使用二项形式的矩阵乘法和加法，直接通过图的相邻矩阵生成。在我们的系列例子中，边 e_1 在节点 n_1 和 n_2 之间，边 e_4 在节点 n_2 和 n_5 之间，在相邻矩阵与其自身的乘积中，位置（1，2）元素构成与位置（2，5）元素的乘积，产生位置（1，5）元素，对应于 n_2 和 n_5 之间的由两条边组成的路径。如果再通过原始相邻矩阵乘以乘积矩阵，则可以得到所有由三条边组成的路径，依此类推。对此，纯数学工作者在确定图中最长路径的长度方面已进行了深入的研究，我们不考虑这些问题，而是将注意力集中到路径将图的"遥远"部件连接起来这种事实。

图4-1中的图容易产生一个问题。它并不是完全通用的，因为它没有显示可能在图中出现的所有情况。具体地说，就是没有给出在路径中节点出现两次的路径。如果有这种情况，则路径就是一个循环（即回路）。通过在节点 n_3 和 n_6 之间加一条边，可产生一个回路。

4.1.5 连接性

路径使我们可以讨论被连接的节点，这是一种强有力的简化工具，对于测试人员非常重要。

定义

节点 n_i 和 n_j 是被连接的，当且仅当它们都在同一条路径上。

"连接性"是一种图的节点集合上的等价关系（请参阅第3章）。为了说明这一点，可以再复习一遍定义等价关系的三个性质：

1. 连接性是自反的，因为每个节点显然都在到其本身长度为0的路径上。
2. 连接性是对称的，由于如果 n_i 和 n_j 在一条路径上，则 n_j 和 n_i 也在同一条路径上
3. 连接性是传递的（请参阅关于长度为2的路径相邻矩阵相乘的讨论）

等价关系引出一个划分（请参阅第3章），因此，可以保证连接性定义图的节点集合上的一个划分。这样就可以定义图的组件了：

定义

图的组件是相连节点的最大集合

等价类中的节点是图的组件 类是最大的，因为等价关系具有传递性 图4-1中的图有两个组件：{ n_1，n_2，n_3，n_4，n_5，n_6}和{n_7}。

4.1.6 压缩图

我们终于可以为测试人员形式化一个重要的简化机制。

定义

给定图G =（V，E），其压缩图通过用压缩节点替代每个组件构成。

开发给定图的压缩图是一种无歧义（即算法的）过程。我们使用相邻矩阵表示路径连通性，再使用等价关系标识组件，这个过程的绝对性质非常重要：给定图的压缩图是惟一的，这意味着所产生的简化代表原始图的一个重要方面。

我们系列例子中的组件是 S_1 = { n_1、n_2，n_3，n_4、n_5，n_6}和 S_2 = {n_7}

在一般（无向）图的压缩图中不能表示边，这有两个原因：

1. 边由个体节点作为端点，而不是节点的集合。（这里我们终于可以区分 n_7 和{n_7}了）。
2. 即使我们没有很好地给出边的定义，忽略了这种差别，那么可能的边仍然意味着来自不同组件的节点是连接的，因此在一条路径上、即在同一个（最大的！）组件中

这对于测试的意义是组件是以重要的方式独立的，因此可以单独测试。

4.1.7 圈数

图的另一个性质对于测试有很深刻的意义，即圈复杂度。

定义

图G的圈数由V（G）= e−n + p给出，其中：

e是G中的边数。

n是G中的节点数。

p是G中的组件数。

V（G）是图中不同区域的个数。我们曾经讨论过向量空间和基本集合概念，结构性测试的一个定理要求使用程序中的基本路径概念，并显示程序图的圈数（请参阅本章末的讨论）是这些基本元素的数量。

我们系列例子的圈数是V（G）= 5−7 + 2 = 0。通过在测试中使用圈复杂度，则（通常）会得到强连接图，将生成较高圈复杂度的图。

4.2 有向图

有向图对一般图稍微做了改进：边有了方向含义。在符号上，无序对偶（n_i, n_j）变成有序对偶<n_i, n_j>，我们说有向边从节点 n_i 到 n_j，而不是在节点之间。

定义

有向图（或框图）D =（V，E）包含：一个节点的有限集合V = {n_1，n_2，…，n_m}，一个边的集合E = {e_1，e_2，…，e_p}，其中每条边e_k = <n_i，n_j>是节点n_i、n_j ∈ V的一个有序对偶。

在有向边e_k = <n_i，n_j>中，n_i是初始（或开始）节点，n_j是终止（或结束）节点。有向图中的边自然地适合很多软件概念：串行行为、命令式程序设计语言、按时间顺序的事件、定义/引用对偶、消息、函数和过程调用，等等。讨论到这里，读者可能会问为什么我们要花费（浪费？）这么多时间在一般图上。一般图和有向图之间的差别与说明式和命令式程序设计语言之间的差别有很强的类比性。在命令式语言（例如COBOL、FORTRAN、Pascal、C、Ada®）中，源语言语句的串行顺序决定编译后的代码执行时间顺序。而说明式语言（例如Prologue）不是这样。对于大多数软件开发人员来说，最常见的说明式例子是定义实体/关系（E/R）模型。在（E/R）模型中，我们把实体作为节点，并把关系表示为边。（如果关系涉及三个或更多实体，则需要引入拥有三个或更多端点的"超边"概念。）E/R模型所产生的图可以更好地解释为一种一般图。好的E/R建模实践压制了有向图所倡导的串行考虑。

在测试以描述式语言编写的程序时，测试人员可用的概念都是一般图中的概念。幸运的是，大部分软件都是采用命令式语言开发的，因此测试人员通常可以具有任意利用有向图的能力。

以下的定义系列大致与一般图的定义平行。我们对大家已经熟悉了的系列例子做一些改动，如图4-2所示。

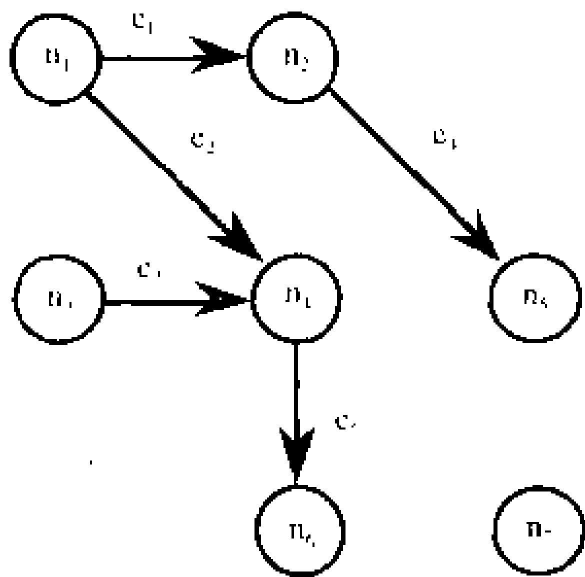


图4-2　一个有向图

我们拥有同样的节点集合V = {n_1，n_2，n_3，n_4，n_5，n_6，n_7}，边集合看起来也相同：

$E = \{e_1, e_2, e_3, e_4, e_5\}$，不同之处是边现在是V中节点的有序对偶：

$$E = \{<n_1, n_2>, <n_1, n_4>, <n_3, n_4>, <n_2, n_5>, <n_4, n_6>\}$$

4.2.1 内度与外度

把一般图中节点的度加以改进以反映方向，如下所示：

定义

有向图中节点的内度，是将该节点作为终止节点的不同边的条数。节点n的内度记做indeg（n）。

有向图中节点的外度，是将该节点作为开始节点的不同边的条数。节点n的外度记做outdeg（n）。

图4-2中有向图的节点具有以下内度和外度：

$$\begin{aligned}
indeg(n_1) &= 0 \quad outdeg(n_1) = 2 \\
indeg(n_2) &= 1 \quad outdeg(n_2) = 1 \\
indeg(n_3) &= 0 \quad outdeg(n_3) = 1 \\
indeg(n_4) &= 2 \quad outdeg(n_4) = 1 \\
indeg(n_5) &= 1 \quad outdeg(n_5) = 0 \\
indeg(n_6) &= 1 \quad outdeg(n_6) = 0 \\
indeg(n_7) &= 0 \quad outdeg(n_7) = 0
\end{aligned}$$

一般图和有向图通过将明显的对应性联系起来的定义关联起来，例如deg（n）= indeg（n）+outdeg（n）。

4.2.2 节点的类型

有向图额外的描述能力使我们能够定义不同种类的节点：

定义

内度为0的节点是源节点。

外度为0的节点是吸收节点。

内度不为0，并且外度不为0的节点是传递节点。

源节点和汇节点构成图的外部边界。如果画出的是背景框图的有向图（根据结构化分析产生的一组数据流图），则外部实体是源节点和汇节点。

在我们的系列例子中，n_1、n_3和n_7是源节点，n_5、n_6和n_7是汇节点，n_2和n_4是传递（又叫做内部）节点。既是源节点又是汇节点的节点是孤立节点。

4.2.3　有向图的相邻矩阵

正如我们所预期的，在边上增加方向，会改变有向图相邻矩阵的定义（关联矩阵也发生变化，但是关联矩阵很少结合有向图使用。）

定义

有m个节点的有向图D =（V，E）的相邻矩阵是一种m × m矩阵：A =（a（i，j）），其中a（i，j）是1，当且仅当从节点i到节点j有一条边，否则该元素为0。

有向图的相邻矩阵不一定是对称的。行的和是节点的外度，列的和是节点的内度。我们系列例子的相邻矩阵是：

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	0	1	0	1	0	0	0
n_2	0	0	0	0	1	0	0
n_3	0	0	0	1	0	0	0
n_4	0	0	0	0	0	1	0
n_5	0	0	0	0	0	0	0
n_6	0	0	0	0	0	0	0
n_7	0	0	0	0	0	0	0

有向图的一种常见使用是记录家庭关系，在这种关系中，兄弟姐妹、堂表兄弟姐妹等都由一个祖先连接，父母、祖父母等都由一个后代连接。现在利用相邻矩阵的表项可显示是否存在有向路径。

4.2.4　路径与半路径

方向为有向图中连接节点的路径带来更精确的含义。作为一种便捷的类比，我们可以通过单向街道和双向街道来考虑这个问题。

定义

（有向）路径是一系列边，使得对于该序列中的所有相邻边对偶e_i、e_j来说，第一条边的终止节点是第二条边的初始节点。

环路是一个在同一个节点上开始和结束的有向路径。

（有向）半路径是一系列边，使得对于该序列中至少有一个相邻边对偶e_i、e_j来说，第一条边的初始节点是第二条边的初始节点，或第一条边的终止节点是第二条边的终止节点。

有向路径有时又叫做链，第9章将使用有向路径概念。我们的系列例子包含以下路径和半路径（没有全部列出）：

从 n_1 到 n_6 的一条路径。

n_1 和n_3 之间的一条半路径。

n_2 和 n_4 之间的一条半路径。

n_5 和 n_6 之间的一条半路径。

4.2.5 可到达性矩阵

当采用有向图对应用程序建模时，我们常常要询问有关能到达特定节点的路径的问题。这是一种极为有用的能力，通过有向图的可到达性矩阵可以提供这种能力。

定义

有m个节点的有向图D =（V，E）的可达性矩阵是一种m × m矩阵R =（r（i，j）），其中r（i，j）是1，当且仅当从节点i到节点j有一条路径，否则该元素为0。

有向图D的可到达性矩阵可以通过相邻矩阵A计算如下：

$$R = I + A + A^2 + A^3 + \cdots + A^k$$

其中k是D最长路径的长度，I是单位矩阵。我们系列例子的可到达性矩阵是：

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	0	1	0	1	1	1	0
n_2	0	0	0	0	1	0	0
n_3	0	0	0	1	0	1	0
n_4	0	0	0	0	0	1	0
n_5	0	0	0	0	0	0	0
n_6	0	0	0	0	0	0	0
n_7	0	0	0	0	0	0	0

可到达性矩阵告诉我们，节点n_2、n_4、n_5和 n_6 可以从 n_1 到达，节点 n_5 可以从 n_2 到达，等等。

4.2.6 n-连接性

一般图的连接性扩展到有向图丰富、具有很高描述性的概念。

定义

有向图中的两个节点 n_i 和 n_j 是：

0-连接，当且仅当 n_i 和 n_j 之间没有路径。

1-连接，当且仅当 n_i 和 n_j 之间有一条半路径，但是没有路径。

2-连接，当且仅当 n_i 和 n_j 之间有一条路径。

3-连接，当且仅当从 n_i 到 n_j 有一条路径，并且从 n_j 到 n_i 有一条路径。

没有其他度的连接性。

需要修改我们的系列例子来显示3-连接性，即从 n_6 到 n_3 增加一条新边 e_8，使得图中包含一个环路。

经过这样的修改，图4-3有以下n-连接性实例（没有全部列出）：

n_1 和 n_7 是0-连接。

n_2 和 n_6 是1-连接。

n_1 和 n_6 是2-连接。

n_3 和 n_6 是3-连接。

类似单向街道，不能从 n_2 走到 n_6。

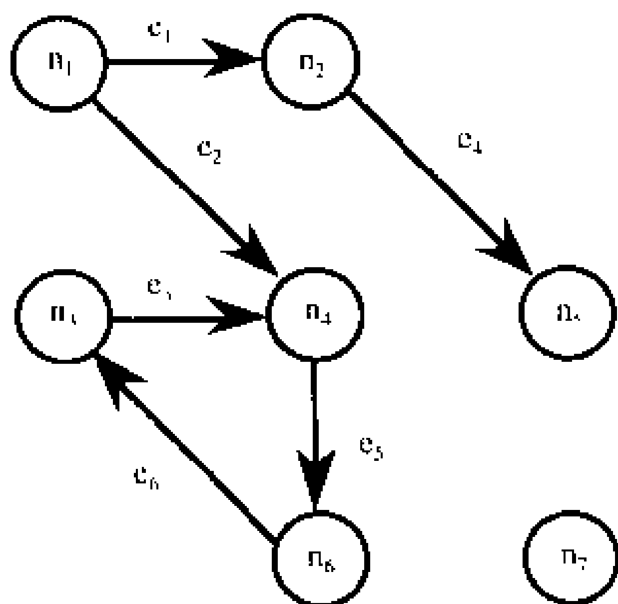


图4-3 带有一个环路的有向图

4.2.7 强组件

继续进行类比。通过n-连接性可以得到两个等价关系：1-连接性产生我们叫做“弱连接”的等价关系，弱连接又产生弱组件。（这些弱组件与一般图的弱组件一样，应该出现这种情况，因为1-连接性实际上不考虑方向。）第二个等价关系基于3-连接性，更有意义。与以前一样，等价关系会得出有向图节点集合上的一个划分，但是压缩图则很不相同。以前是0-、1-或2-连接的节点原封不动，3-连接节点变成强组件。

定义

有向图的强组件是3-连接节点的最大集合。

在我们经过修改的例子中，强组件是集合$\{n_3, n_4, n_6\}$和$\{n_7\}$，这个经过修改的例子的压缩图如图4-4所示。

强组件使我们能够通过清除循环和孤立节点来简化有向图。虽然这种简化不如对一般图简化那样显著，但是它确实可以解决一个主要测试问题。请注意，有向图的压缩图永远不会包含环路。（如果包含环路，则环路会被划分的最大化特性压缩掉。）这些图有特殊的名称：有向无环图，有时又叫做DAG。

有关结构性测试的很多论文都相当清晰地说明相对简单的程序会如何拥有数百万不同的执行路径。这些讨论是要说明彻底测试是那样地耗费资源。造成执行路径很多的原因是嵌套环路。压缩图可以消除环路（或至少将环路压缩为一个节点），因此，可以利用压缩图作

为对付否则在算法上行不通的问题的一种简化策略。

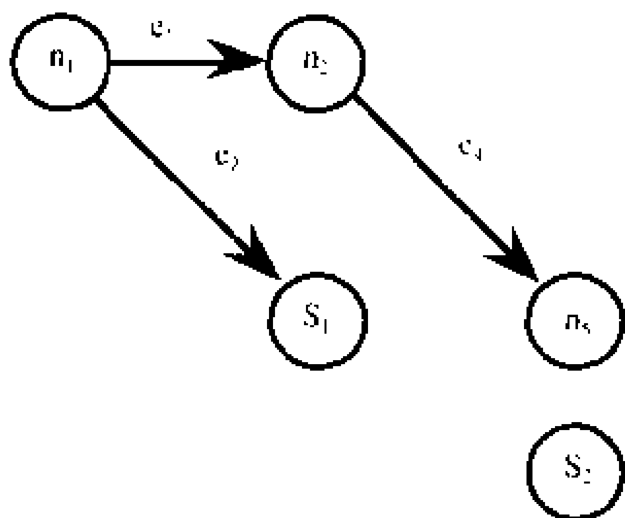


图4-4 图4-3所示有向图的压缩图

4.3 用于测试的图

本章的最后介绍被广泛用于测试的四种特殊的图。第一种是程序图，主要用于单元测试层次，其他三种图分别是，有限状态机、状态图和Petri网，这三种图都最适合用来描述系统级行为，尽管也可以用于较低层次的测试。

4.3.1 程序图

在本章的开始，我们曾经说要避免对图论定义进行解释，解释要留到结合后面的应用程序进行。这里我们给出图论在软件测试中的最常见的使用，即程序图。为了更好地联系现有的测试文献，我们先给出传统定义，然后给出经过改进的定义。

定义

给定一个采用命令式程序设计语言编写的程序，其程序图是一种有向图，其中：

1.（传统定义）

节点是程序语句，边表示控制流（从节点i到节点j有一条边，当且仅当对应节点j的语句可以立即在节点i对应的语句之后执行）。

2.（经过改进的定义）

节点要么是整个语句，要么是语句的一部分，边表示控制流（从节点i到节点j有一条边，当且仅当对应节点j的语句或语句的一部分，可以立即在节点i对应的语句或语句的一部分之后执行）。

总是说“语句或语句的一部分”很罗嗦，因此我们约定语句的一部分也可以是整个语句。程序的有向图公式化能够非常准确地描述程序的测试方面的问题。首先，这种公式化

和结构化程序设计的客体之间有了令人满意的联系。基本结构化程序设计的构造（串行、选择和反复）都具有清晰的有向图，如图4-5所示。

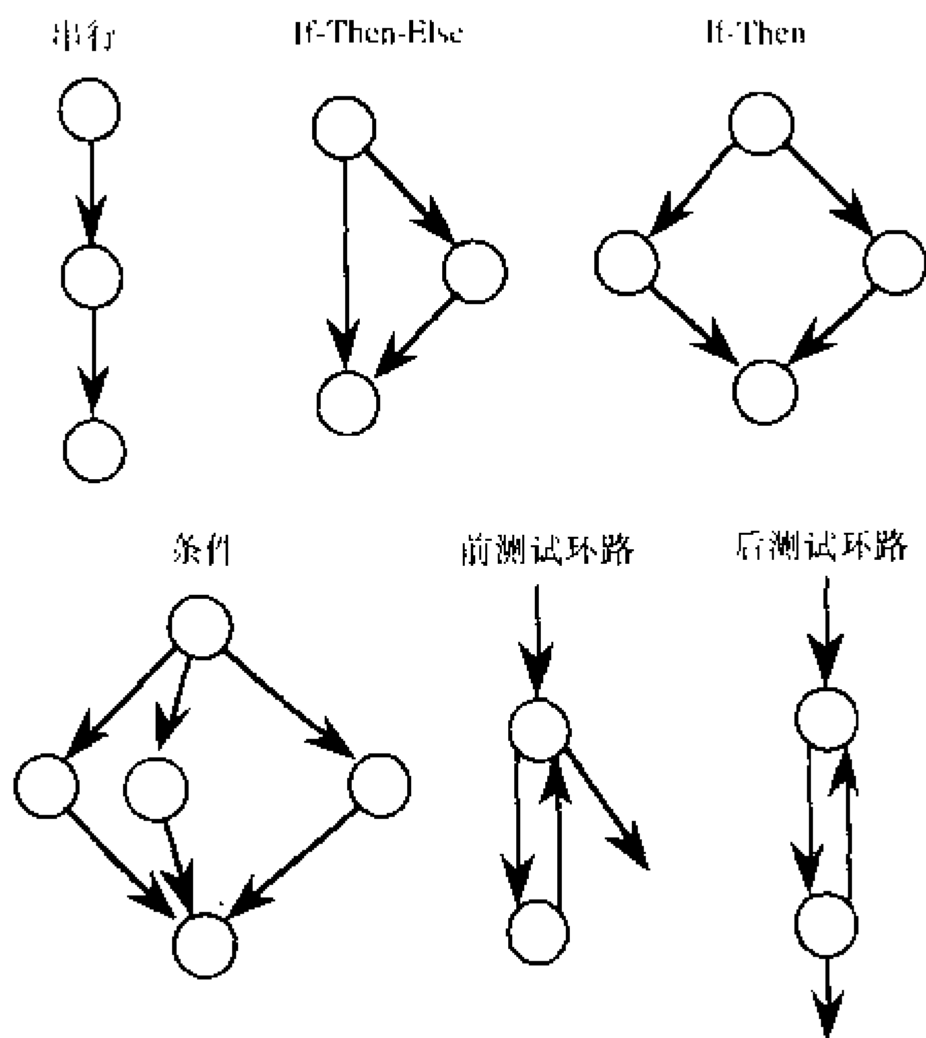


图4-5 结构化程序设计构造的有向图

当把这些构造用于结构化程序中，对应的图要么是嵌套的，要么是压缩的。但单入口、单出口的评判准则要求在程序图中有惟一的源节点和惟一的汇节点。事实上，老的（非结构化的）"空心节点"，会产生非常复杂的程序图。例如，GOTO语句会引入边，当GOTO语句用于跳入或跳出循环时，所产生的程序图甚至更复杂。在这个方面最早进行研究的学者之一是Thomas McCabe，他使图的圈数作为程序复杂度而成为普遍采用的指标（McCabe，1976）。当执行程序时，所执行的语句构成程序图中的一条路径。环路和判断大大增加了可能的路径数，因此需要针对测试进行类似的降低。

程序图的一个问题是如何处理非可执行语句，例如注释和数据说明语句。最简单的回答是不考虑这些语句。第二个问题与拓扑结构可能和语义可能的路径之间的差别有关。第三部分将更详细地讨论这个问题。

4.3.2 有限状态机

有限状态机已经成为需求规格说明的一种相当标准的表示方法。所有结构化分析的实时扩展，都要使用某种形式的有限状态机，并且几乎所有形式的面向对象分析也都要使用有限状态机。有限状态机是一种有向图，其中状态是节点，转移是边。源状态和吸收状态是

初始节点和终止节点，路径被建模为通路，等等。大多数有限状态机表示方法都要为边（转移）增加信息，以指示转移的原因和作为转移的结果要发生的行动。

图4-6是一个简单的自动柜员机（SATM）系统中，用于个人标识编号（PIN）尝试部分的有限状态机。这种机器包含5个状态（空闲、等待第一次PIN尝试，等等）和8个用边表示的转移。转移上的标签所遵循的规则是，“分子”是引起转移的事件，“分母”是与该转移关联的行为。事件是必须给出的，即转移不能无原由地发生，但是可以没有行动。有限状态机是表示可能发生的各种事件，以及其发生产生的不同结果的简单方法。例如，在SATM系统的PIN输入部分中，客户有三次机会输入正确的PIN数字。如果第一次就输入正确的PIN，则SATM系统会采取输出行为，显示屏幕5（提示客户选择事务类型）。如果输入的PIN不正确，则机器会进入一个不同的状态，等待第二次PIN尝试。请注意，从等待第二次PIN尝试状态转移时的事件和行为，与第一次转移的事件和行为相同。这就是有限状态机保留过去事件历史的方式。

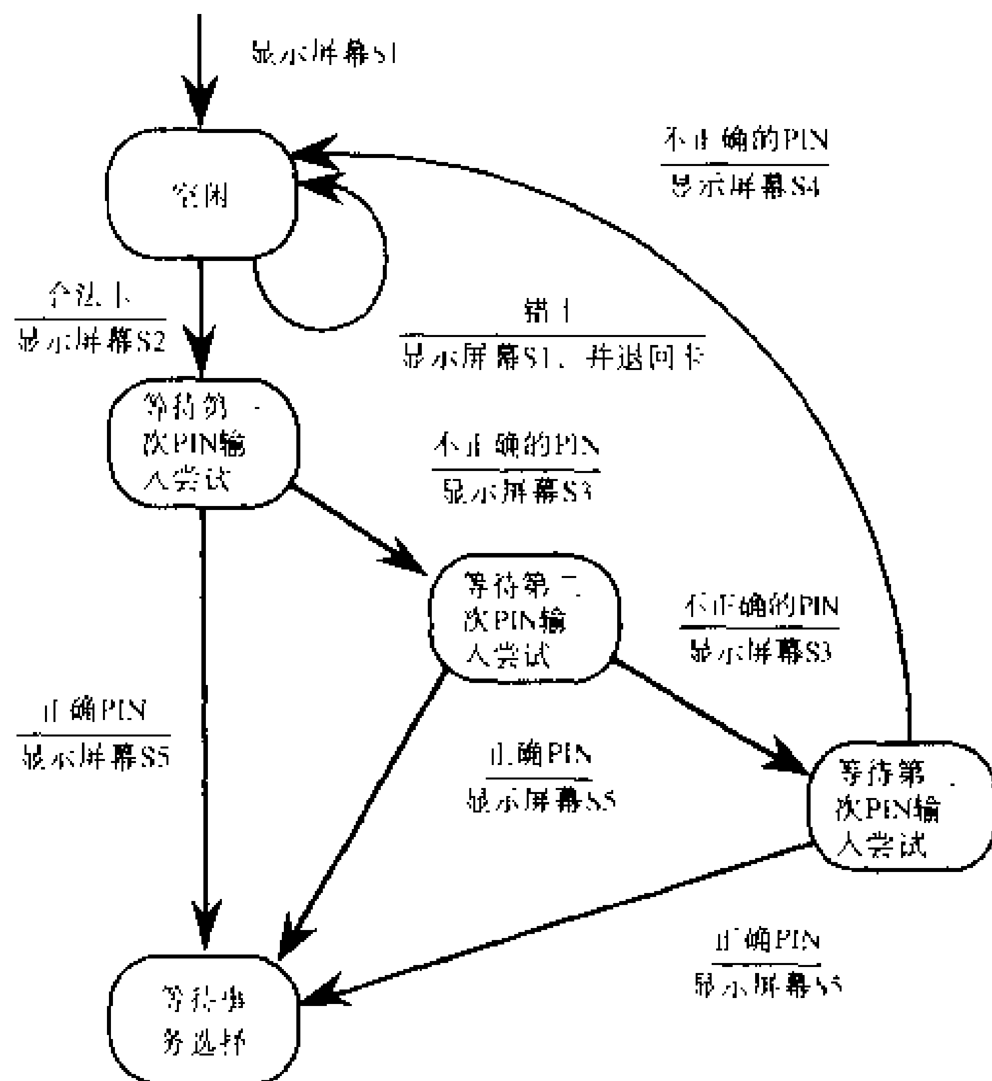


图4-6 用于PIN尝试的有限状态机

有限状态机可以执行，但是首先要定义一些约定。一条约定是活动状态的概念。我们说系统“处于”一定状态，如果系统被建模为有限状态机，则活动状态是指“我们所处”的状态。另一条约定是，有限状态机可能有一个初始状态，即最初进入该有限状态机时是活动的状态。（在图4-6中，空闲状态是初始状态，由没有从其他地方来的转移表示。最终状态由没有转出的转移表示。）在任何时间，一次只能有一个状态是活动的。我们还可以把转移看

做是瞬间发生，引起转移的事件也是一次发生一个。为了执行有限状态机，我们要从初始状态开始，并提供引起状态转换的事件序列。每次事件发生时，转移都会改变活动状态，并发生新的事件。通过这种方式，一系列事件会选择通过有限状态机的状态路径（也就是与之等效的转移）。

4.3.3 Petri网

Petri网是Carl Adam Petri于1963年完成的博士论文的题目。今天，Petri网已被接受为原型和涉及并发和分布式处理应用程序的模型。Petri网是一种特殊形式的有向图：一种双向有向图。（双向图有两个节点集合 V_1 和 V_2，以及一个边集合E，要求每条边在 V_1 和 V_2 集合之一中有自己的初始节点，在另一个节点集合中有自己的终止节点。）在Petri网中，集合之一被叫做“地点”，另一个集合被叫做“转移”。这些集合一般分别记做P和T。地点是要输入的节点，转移是要输出的节点。输入和输出关系是函数，通常记做In和Out，如以下定义所示

定义

Petri网是一种双向有向图（P，T，In，Out），其中，P和T是不相交的节点集合，In和Out是边集合，$In \subseteq P \times T$，$Out \subseteq T \times P$。

对于图4-7给出的示例Petri网，集合P、T、In和Out是：

$P = \{p_1, p_2, p_3, p_4, p_5\}$

$T = \{t_1, t_2, t_3\}$

$In = \{<p_1, t_1>, <p_5, t_1>, <p_5, t_3>, <p_2, t_3>, <p_3, t_2>\}$

$Out = \{<t_1, p_3>, <t_2, p_4>, <t_3, p_4>\}$

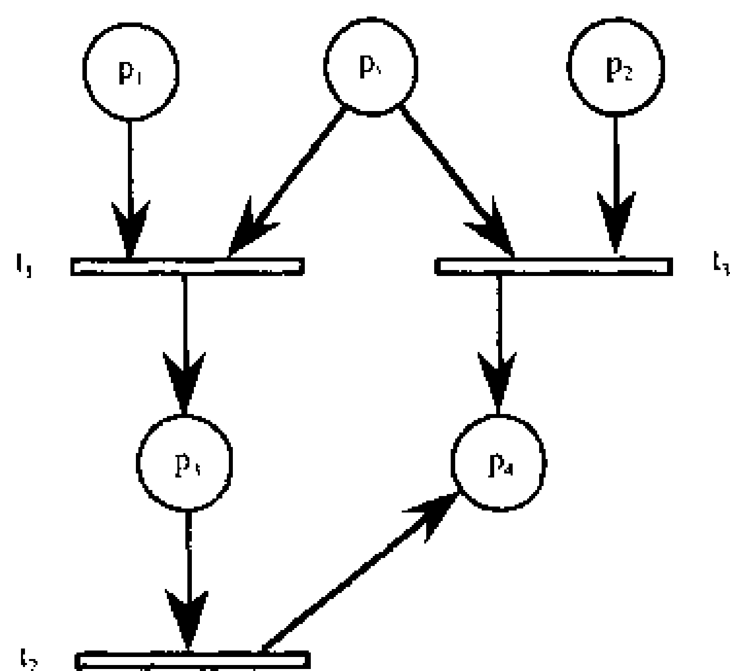


图4-7 Petri网

Petri网能够以比有限状态机更有意义的方式执行。以下定义使Petri网能够执行。

定义

有标记的Petri网是一种5元组（P、T、In，Out，M），其中（P、T、In，Out）是一个Petri网，M是从地点到正整数的映射集合。

集合M叫做Petri网的标记集合。M中的元素是n元组，其中n是集合P中的地点个数。对于图4-7所示的Petri网，集合M包含形式为<n_1，n_2，n_3，n_4，n_5>的元素，其中n是指向被称做"位于"该地点的记号个数的指针。记号是抽象，可以解释为建模环境。例如，记号可以指向地点被使用过的次数，或地点中的事物个数，或地点是否为真。图4-8给出了一个有标记的Petri网。

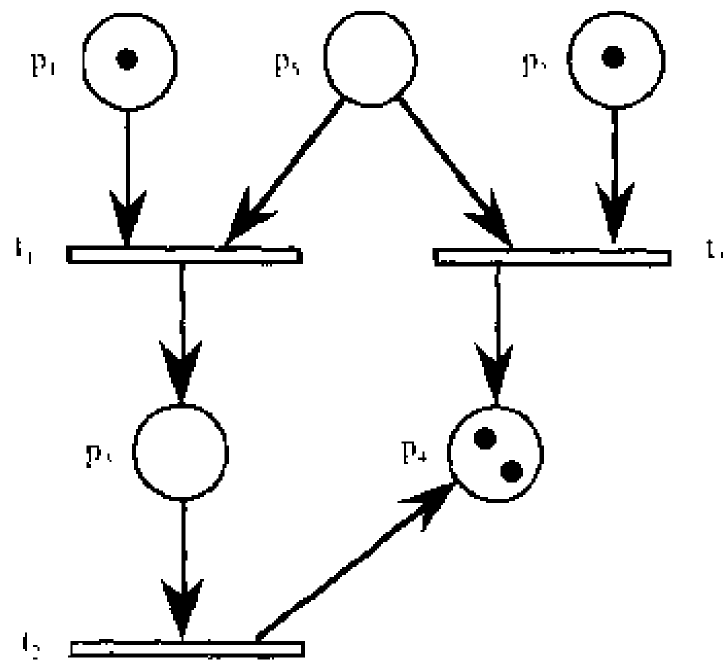


图4-8 有标记的Petri网

图4-8所示有标记的Petri网的标记元组是<1，1，0，2，0>。我们需要使用记号概念进行两个基本定义。

定义

转移可以在Petri网中发生，如果在其所有输入地点中至少有一个记号。

图4-8所示Petri网中没有能够进行的转移。如果将一个记号放入地点p_3，则转移t_2就可以发生。

定义

当触发Petri网一个转移时，要从其每个输入地点删除一个记号，并在其每个删除地点增加一个记号。

在图4-9中，转移t_2在上面的网络中被允许发生，在下面的网络中触发。

图4-9所示网络的标记集合包含两个元组，第一个元组显示当t_2在上面的网络中被允许发生的情况，第二个元组表示该网络在t_2触发后的情况。

$$M = \{<1, 1, 1, 2, 0>, <1, 1, 0, 3, 0>\}$$

通过转移触发，集合可以被创建或销毁。在特殊条件下，网络中的记号总数永远不变，

这种网络叫做守恒网络。我们通常不关心记号守恒问题。标记使我们能够与执行有限状态机非常类似地执行Petri网。（已经证明，有限状态机是Petri网的一种特例。）假设图4-7有一种不同的标记，在这种标记中，地点 p_1、p_2 和 p_5 都被标记。通过这种标记，转移 t_1 和 t_3 都被允许发生。如果选择触发转移 t_1，则地点 p_5 的记号要被删除，转移 t_3 不再被允许发生。类似地，如果选择触发转移 t_3，就会禁止 t_1 发生。这种模式叫做Petri网冲突。更具体地说，我们说转移 t_1 和 t_3 关于地点 p_5 冲突。Petri网冲突展示了两个转移之间交互的一种有意义的形式，第15章还要讨论这种交互（以及其他交互）。

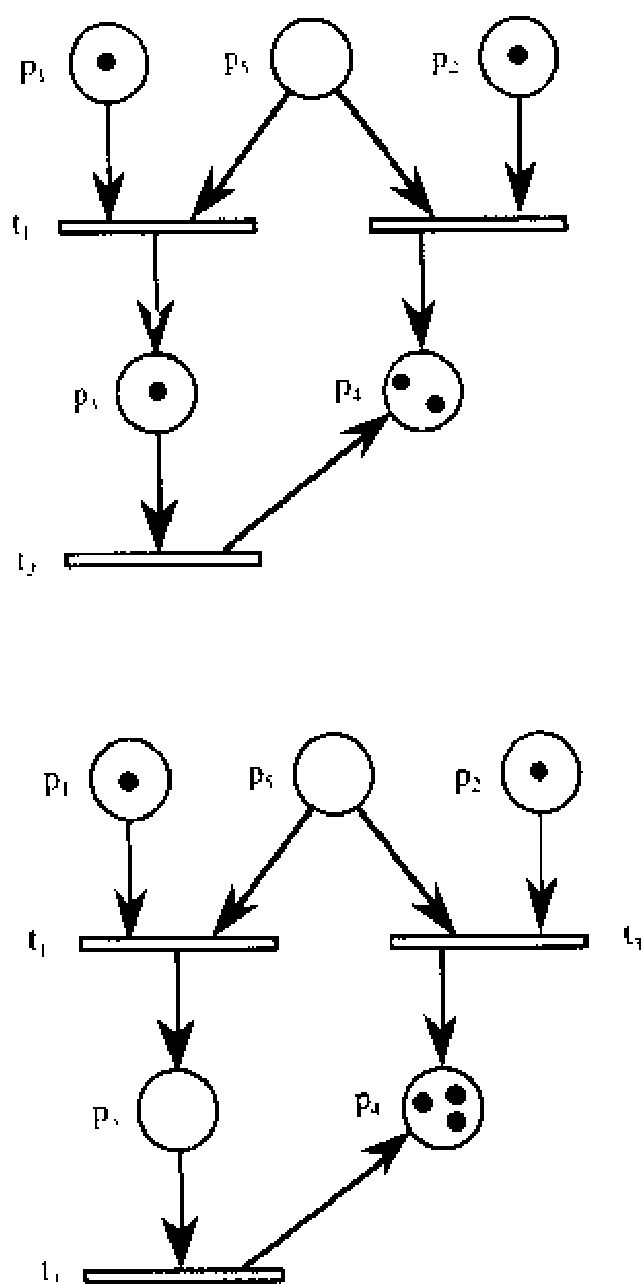


图4-9 触发 t_2 之前和之后

4.3.4 事件驱动的Petri网

对基本Petri网的两处稍做修改就可以构成事件驱动的Petri网（EDPN）。第一处修改使Petri网能够更接近地表示事件驱动的系统，第二处修改解决表示事件静止的Petri网标记问题，这是面向对象应用程序的一种重要概念。通过这两处修改，可产生软件需求的一种高效的操作视图，在其他地点这种视图叫做OSD（“操作软件开发”的英文缩写）网（Jorgensen，1989）。

定义

EDPN是一种多向图（P，D，S，In，Out），包括三个节点集合P、D和S，以及两个映射集合In和Out，其中：

P是端口事件的集合。
D是数据地点的集合。
S是转移的集合。
In是（P∪D）×S的有序对偶集合。
Out是S×（P∪D）的有序对偶集合。

EDPN表示第14章定义的五种基本系统构造中的四种，只有设备除外。转移的S集合对应一般Petri网转移，被解释为行为。两种地点，即端口事件和数据地点，是由输入和输出函数In和Out定义的S中转移的输入和输出。线索是S中的转移序列，因此我们总是能够通过线索中的转移输入和输出，构造线索的输入和输出。EDPN的图形表示与一般Petri网很相似，惟一的差别是使用三角表示端口事件地点。图4-10中的EDPN有四个转移，即s_7、s_8、s_9和s_{10}；两个输入事件，即p_3和p_4；三个数据地点，即d_5、d_6和d_7。没有端口输出事件。

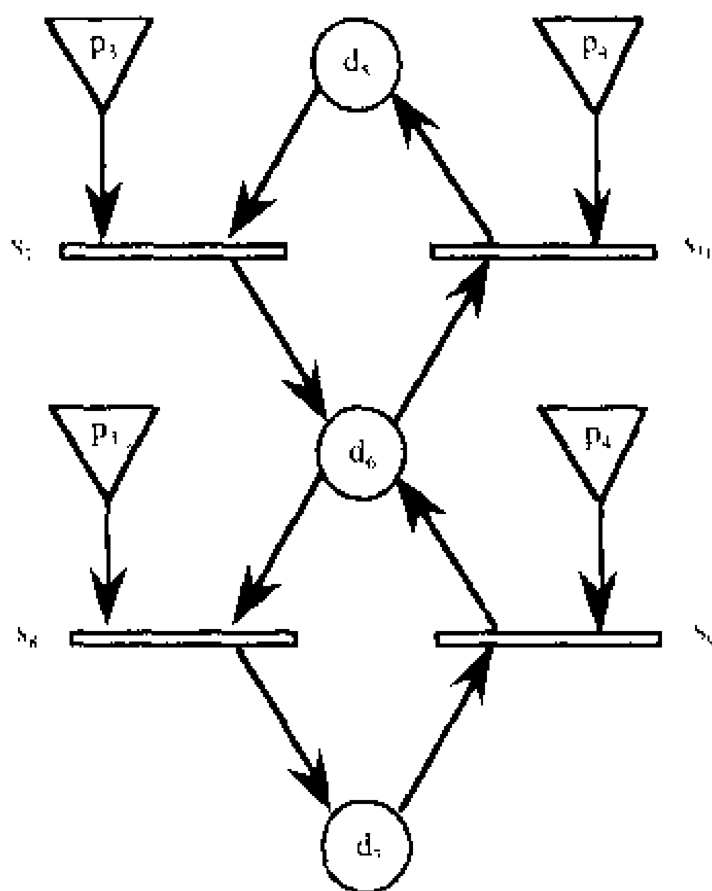


图4-10 EDPN

这个EDPN对应于第15章介绍土星牌挡风玻璃雨刷部分开发的有限状态机（请参阅图15-10）。这个网络的组件在表4-1中描述。

表4-1 图4-10中的EDPN元素

元素	类型	描述
p_3	端口输入事件	顺时针旋转
p_4	端口输入事件	逆时针旋转
d_5	数据地点	转到位置1
d_6	数据地点	转到位置2
d_7	数据地点	转到位置3
s_7	转换	状态转换：d_5到d_6
s_8	转换	状态转换：d_6到d_7
s_9	转换	状态转换：d_7到d_6
s_{10}	转换	状态转换：d_6到d_5

为EDPN做标记更复杂，因为我们要求能够处理事件静止。

定义

EDPN（P，D，S，In，Out）的一个标记M，是p元组的一个序列$M = <m_1, m_2, \cdots>$，其中$p = k + n$，k和n是集合P和D中的元素个数，p元组中的个体项表示事件或数据地点中的记号个数。

根据约定，我们把数据地点放在开头，然后是输入事件地点，然后是输出事件地点。EDPN可以拥有任意数量的标记，每个标记都对应网络的一个执行。表4-2给出了图4-10中EDPN的一个示例标记。

表4-2 图4-10中EDPN的一个标记

元组	$(p_3, p_4, d_5, d_6, d_7)$	描述
m_1	(0, 0, 1, 0, 0)	初始条件，处于状态d_5
m_2	(1, 0, 1, 0, 0)	p_3发生
m_3	(0, 0, 0, 1, 0)	处于状态d_6
m_4	(1, 0, 0, 1, 0)	p_3发生
m_5	(0, 0, 0, 0, 1)	处于状态d_7
m_6	(0, 1, 0, 0, 1)	p_4发生
m_7	(0, 0, 0, 1, 0)	处于状态d_6

EDPN中的允许和触发转移规则，正好与传统Petri网的允许和触发转移规则类似。如果在每个输入地点至少有一个记号，则允许转移发生；当所允许的转移被触发时，要从每个输入地点删除一个记号，在其每个输出地点放入一个记号。表4-3给出了允许发生和被触发的转移。

表4-3 表4-2中允许发生和被触发的转移

元组	$(p_3, p_4, d_5, d_6, d_7)$	描述
m_1	(0, 0, 1, 0, 0)	没有允许发生的转移
m_2	(1, 0, 1, 0, 0)	s_7允许发生，s_7被触发
m_3	(0, 0, 0, 1, 0)	没有允许发生的转移
m_4	(1, 0, 0, 1, 0)	s_8允许发生，s_8被触发
m_5	(0, 0, 0, 0, 1)	没有允许发生的转移
m_6	(0, 1, 0, 0, 1)	s_9允许发生，s_9被触发
m_7	(0, 0, 0, 1, 0)	没有允许发生的转移

EDPN和传统Petri网的一个重要差别，是可以通过在端口输入事件地点中创建一个记号分解事件静止。在传统Petri网中，如果没有允许发生的转移，则我们说该网络死锁。在EDPN中，如果没有允许发生的转移，则网络是处于事件静止中。（当然，如果没有事件发生，这种情况与死锁相同。）在表4-3中，事件静止在m_1、m_3、m_6和m_7上出现了4次。

标记中的个体成员可以看做是在分立的时间点上执行EDPN的快照，这些成员也可以叫做时间步骤、p元组或标记向量。这使我们可以把时间看做是一种使我们能够分辨"之前"和"之后"的排序。如果我们将瞬态时间作为端口事件、数据地点和转移的一个属性，则可以得到线索行为的更加清晰的描述。这样做的缺点是，如何在端口输出事件地点处理记号。端口输出地点总是有外度＝0。在一般Petri网中，记号不能从具有0外度的地点删除。如果端口输出事件地点中的记号持续存在，则说明该事件不确定地发生。同样，时间属性可以解决这个问题，这一次我们需要被标记输出事件的一段持续时间（另一种可能是在一个时间步之后，从被标记输出事件地点输出记号，这样做也很有效）。

4.3.5 状态图

David Harel在开发状态图表示法时有两个目标：他要将维恩图描述层次结构的能力以及有向图描述连接性的能力结合在一起，开发出一种可视化标示法（Harel，1998）。结合到一起，这些能力为一般有限状态机的"状态爆炸"问题提供了一种理想的答案。所产生的结果是高度精巧和非常精确的标记，能够由商业化的CASE工具提供支持，著名的工具有i-Logix公司的StateMate系统。状态图现在被Rational公司选为统一建模语言（UML）的控制模型。（详情请访问www.rational.com。）

Harel使用与方法无关的术语"团点"表示状态图的基本构件块。团点可以像维恩图显示集合包含那样地包含其他团点。团点还可以像在有向图中连接节点一样地通过边连接其他团点。在图4-11中，团点A包含两个团点（B和C），通过边连接。团点A通过边与团点D连接。

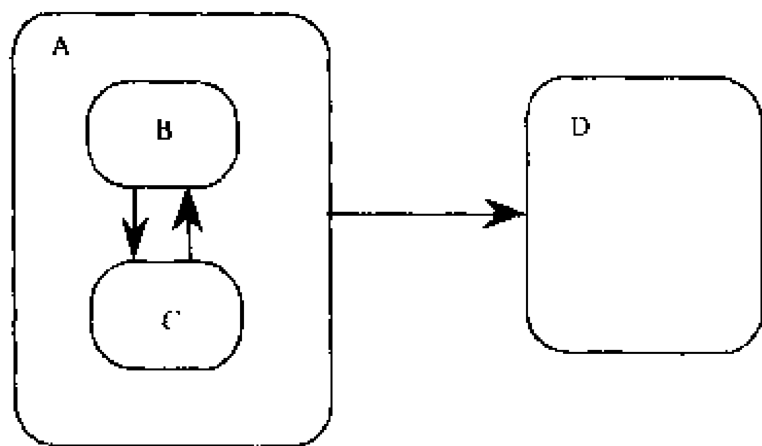


图4-11 状态图中的团点

根据Harel的意图，我们可以把团点解释为状态，把边解释为转移。完整的状态图支持一种精制的语言，定义转移如何发生和什么时候发生（这种语言的培训课程需要整整一周，因此这里只能简单介绍一下）。状态图与一般有限状态机相比，能够以更精细的方式运行。执行状态图需要使用与Petri网标记类似的概念。状态图的"初始状态"由没有源状态的边表示

当状态嵌入在其他状态内部时，使用同样的方式显示低层初始状态．在图4-12中，状态A是初始状态，当进入到这个状态时，也进入低层状态B．当进入某个状态时，我们可以认为该状态是活动的，这可与Petri网中的被标记地点类比。（状态图工具采用色彩表示哪个状态是活动的，并等效于Petri网中的标记地点。）图4-12中有一些微妙的地方，从状态A转移到状态D初看起来是有歧义的，因为它没有区分状态B和C。约定是，边必须开始和结束于状态的外围，如果状态包含子状态，就像图中的A一样，边会“引用”所有的子状态．因此，从A到D的边意味着转移可以从状态B或从状态C发生。如果有从状态D到状态A的边，如图4-13所示，则用B来表示初始状态这个事实，意味着转移实际上是从状态D到状态B．这种约定可以大大减缓有限状态机向“空心代码”发展的趋势。

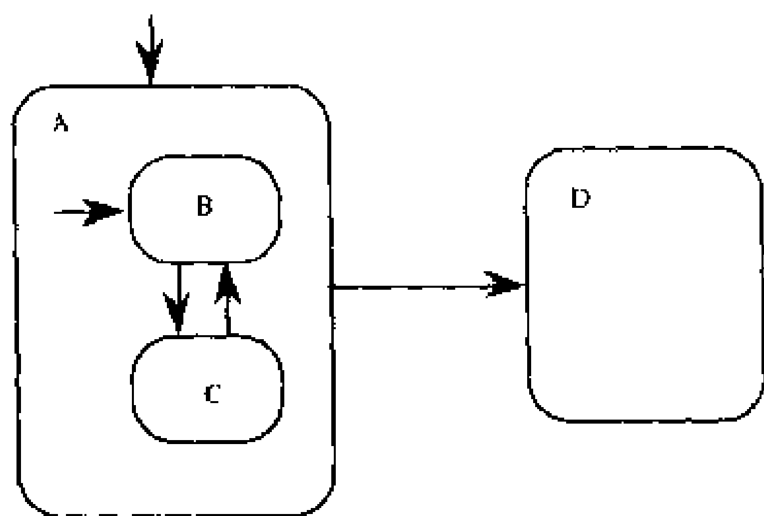


图4-12 状态图中的初始状态

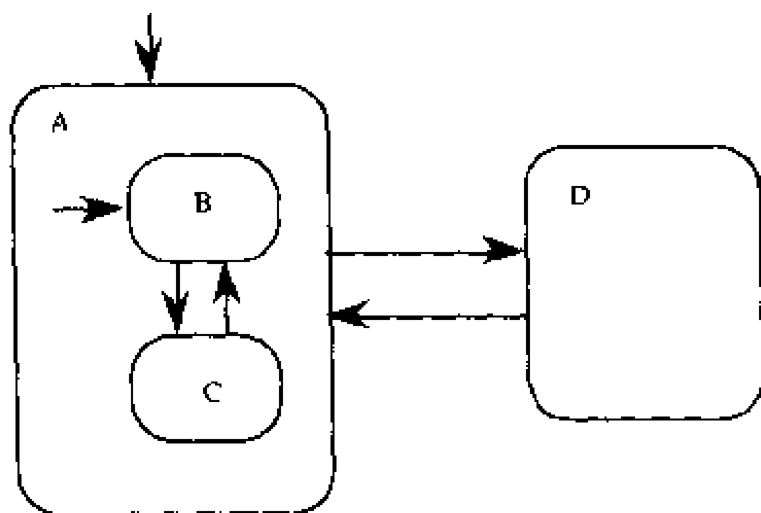


图4-13 进入子状态的默认入口

我们要讨论的状态图的最后一个特性，是并发状态图概念．图4-14中状态D的虚线用于表示状态D实际上引用两个并发状态E和F。（Harel的约定是将状态标签D移到该状态周边的矩形标号上。）虽然这里没有显示出来，但是我们可以把E和F想像为并发执行的平行机器．由于从状态A出来的边在状态D的周边终止，因此当转移发生时，机器E和F都是活动的（用Petri网的术语说，也就是被标记的）

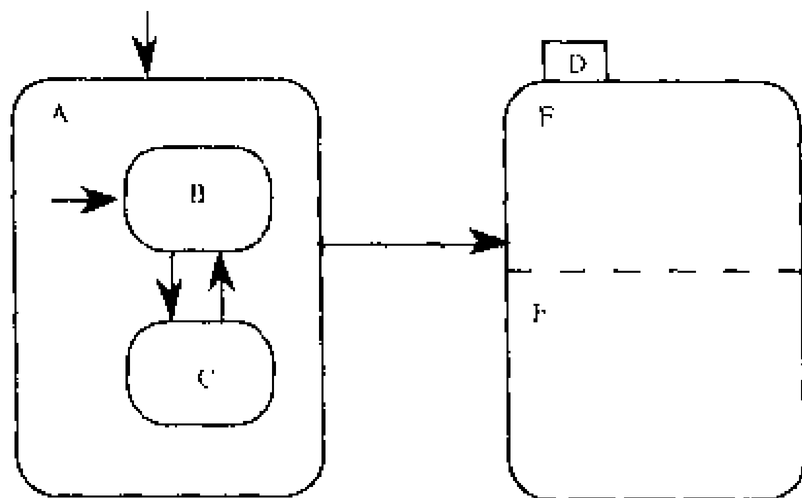


图4-14 并发状态

4.4 参考文献

Harel, David, On visual formalisms, *Communications of the ACM*, Vol. 31, No. 5, pp. 514–530, May, 1988

4.5 练习

1. 请为图中的路径长度提议一个定义。

2. 在图4-1中的图，如果在节点 n_5 和 n_6 之间增加一条边，创建的是什么环路？

3. 请说明3-连接性是有向图节点上的等价关系。

4. 计算图4-5中的每个结构式程序设计构造的圈复杂度。

5. 通过在图4-3中的有向图中增加节点和边，得到如图4-15所示的有向图。请计算每个新有向图的圈复杂度，并解释这些变化怎样影响复杂度。

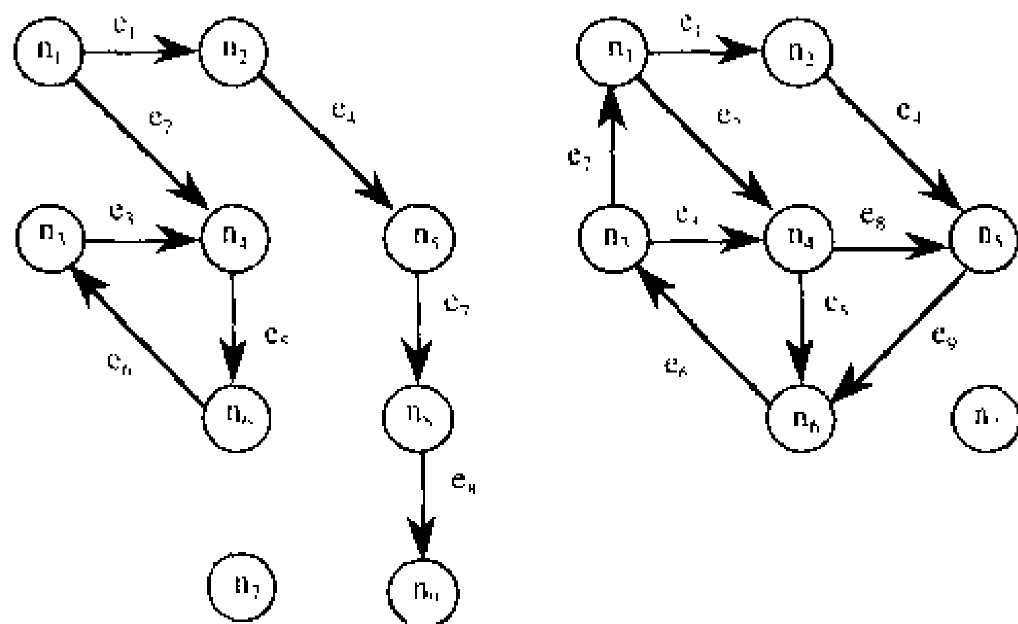


图4-15 练习5用图

6. 假设我们做出一张图，节点表示人，边表示某种形式的社会交互关系，例如“谈话”或“熟悉”。请找出对应社会概念，例如很著名、小圈子和隐居的图论中概念。

第二部分

功能性测试

第5章 边界值测试

在第3章中，我们看到函数将从一个集合（函数的定义域）的值映射到另一个集合（函数的值域）的值上，定义域和值域可以是其他集合的叉积。任何程序都可以看做是一个函数，程序的输入构成函数的定义域，程序的输出构成函数的值域。输入定义域测试是最著名的功能性测试手段。本章和后面两章将讨论如何运用程序的函数特性，为程序标识测试用例。在历史上，这种形式测试的重点是在输入定义域，但是将很多这类手段应用于开发基于值域的测试用例，常常是很好的补充。

5.1 边界值分析

为了便于理解，以下讨论涉及有两个变量 x_1 和 x_2 的函数F。如果函数F实现为一个程序，则输入两个变量 x_1 和 x_2 会有一些（可能未规定）边界：

$$a \leqslant x_1 \leqslant b$$

$$c \leqslant x_2 \leqslant d$$

但是，区间[a，b]和[c，d]是 x_1 和 x_2 的值域，因此我们立即就有一个过多使用的术语其含义根据语境总是很清楚的。强类型语言（例如Ada和Pascal）允许显式地定义这种变量值域。事实上，采用强类型的部分历史原因，就是要防止程序员出现某些类型的错误，这些错误会导致通过边界值测试很容易发现的缺陷。其他语言（例如COBOL、FORTRAN和C）不是强类型语言，因此边界值测试更适用于采用这些语言编写的程序代码。函数F的输入空间（定义域）如图5-1所示。带阴影矩形中的任何点都是函数F的有效输入。

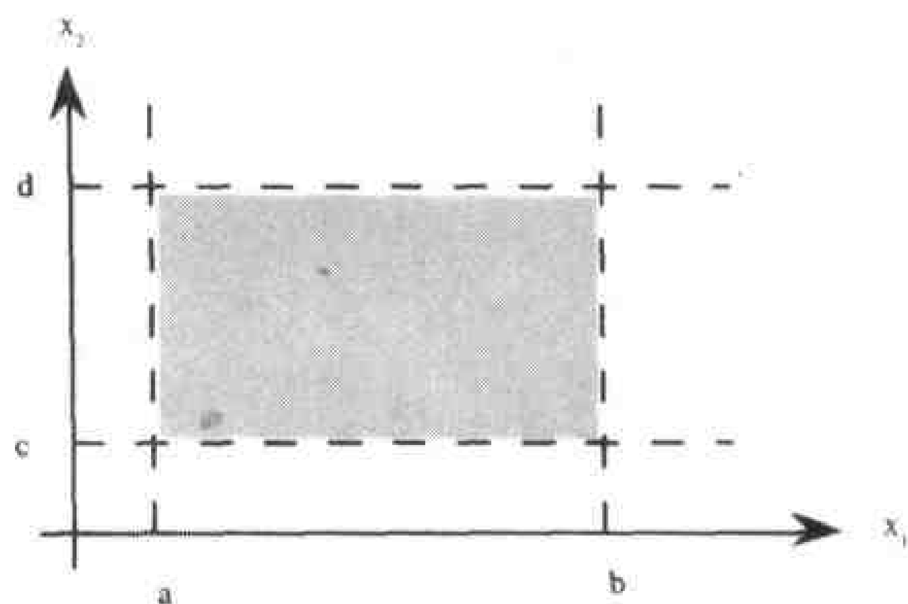


图5-1 两变量函数的输入定义域

边界值分析关注的是输入空间的边界，以标识测试用例。边界值测试背后的基本原理是错误更可能出现在输入变量的极值附近。例如，循环条件可能在应该测试≤时测试了<，并且计数器常常“少记一次”。打印本书手稿的桌面出版程序有一个很有意思的边界值问题。这个程序使用两个文本显示模式：一种模式通过虚线指示新页，另一种模式显示页影像，说明文本在页面中的位置。如果光标位于一页的最后一行，并增加新文本，就会出现一种异常：在第一种模式下，新行被显示出来，虚线（换页）被调整。但是在页面显示模式下，新文本被丢失，在两个页面上都没有显示出来。美国陆军（CECOM）对其软件进行了研究，令人吃惊地发现，大量缺陷都是边界值缺陷。

边界值分析的基本思想是使用在最小值、略高于最小值、正常值、略低于最大值和最大值处取输入变量值。一种商业化测试工具（叫做T）可以为恰当描述的程序生成这类测试用例。这个工具成功地与两个流行的前端CASE工具（Cadre System公司的Teamwork和Aonix公司的Software through Pictures）集成起来。T工具把这些值叫做min、min+、nom、max-和max，这里我们也使用这种约定。

边界值分析的下一个部分基于一种关键假设，在可靠性理论叫做“单缺陷”假设。这种假设是说，失效极少是由两个（或多个）缺陷的同时发生引起的。因此，边界值分析测试用例的获得，通过使所有变量取正常值，只使一个变量取极值。我们的两变量函数F（如图5-2所示）的边界值分析测试用例是：

$$\{<x_{1nom}, x_{2min}>, <x_{1nom}, x_{2min+}>, <x_{1nom}, x_{2nom}>, <x_{1nom}, x_{2max}>, <x_{1nom}, x_{2max-}>, <x_{1min}, x_{2nom}>, <x_{1min+}, x_{2nom}>, <x_{1nom}, x_{2nom}>, <x_{1max}, x_{2nom}>, <x_{1max-}, x_{2nom}>\}$$

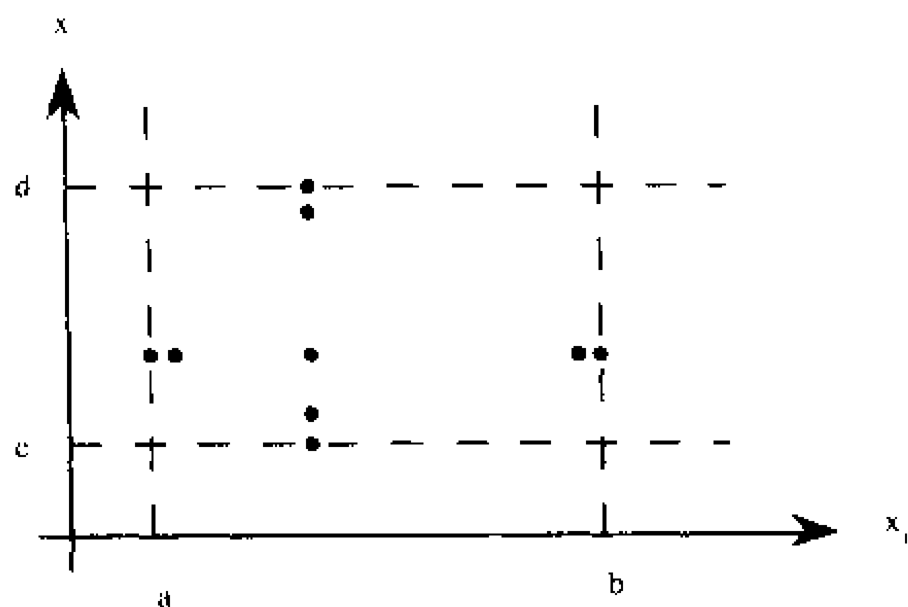


图5-2　两变量函数边界值分析测试用例

5.1.1　归纳边界值分析

基本边界值分析手段可以用两种方式归纳：通过变量数量和通过值域的种类。归纳变量

数量很容易：如果有一个n变量函数，使除一个以外的所有变量取正常值，使剩余的那个变量取最小值、略高于最小值、正常值、略低于最大值和最大值，对每个变量都重复进行这样，对于一个n变量函数，边界值分析会产生4n + 1个测试用例。

归纳值域取决于变量本身的性质（或更准确地说是类型）。例如，对于NextDate函数，我们有月份、日期和年对应的变量。采用类似于FORTRAN的语言，很有可能对这些变量编码，使得1月对应1，2月对应2，等等。采用支持用户定义类型的语言（例如Pascal或Ada），可以把变量“月份”定义为枚举类型{1月，2月，...，12月}。不管采用什么语言，最小值、略高于最小值、正常值、略低于最大值和最大值根据语境可以很清楚地确定。如果变量具有离散、有界值，例如佣金问题中的变量，则最小值、略高于最小值、正常值、略低于最大值和最大值也可以很容易地确定。如果没有显式地给出边界，例如三角形问题，通常必须创建一种“人工”边界。边长的最低值显然是1（负的边长很傻），但是上限怎样确定呢？在默认情况下，最大可表示整数（在某些语言中叫做MAXINT）是一种可能，也可以任意规定上限，例如200或2000。

边界值分析对布尔变量没有什么意义，极值是TRUE和FALSE，但是其余三个值不明确。第7章将要介绍，布尔变量可以进行基于决策表的测试。逻辑变量也代表一种边界值分析。在ATM例子中，客户的PIN是一个逻辑变量，事务处理类型也是逻辑变量（存款，取款，查询）。对于这种变量可以“遍历”边界值分析测试，但是这样做与“测试人员的直觉”并不非常吻合。

5.1.2 边界值分析的局限性

如果被测程序是多个独立变量的函数，这些变量受物理量的限制，则很适合边界值分析。这里的关键词是“独立”和“物理量”。简单地看一下NextDate的边界值分析测试用例（请参阅第5.5节），就会发现这些测试用例是不充分的。例如，没怎么强调2月和闰年。这里的真正问题是，月份、日期和年变量之间存在有意思的依赖关系。边界值分析假设变量是完全独立的。即便如此，边界值分析也能够捕获月末和年末缺陷。边界值分析测试用例通过引用物理量的边界独立变量极值导出，不考虑函数的性质，也不考虑变量的语义含义。我们把边界值分析测试用例看做是初步的，因为这些测试用例的获得基本没有利用理解和想像。一分耕耘，一分收获，付出多少，收获多少。

物理量准则也很重要。如果变量引用某个物理量，例如温度、压力、空气速度、迎角、负载等，物理边界极为重要。（举一个这方面的很有意思的例子，菲尼克斯的航空港国际机场1992年6月26日被迫关闭，因为空气温度达到122°F。飞行员在起飞之前不能设置特定设备：该设备能够接受的最大空气温度是120°F。）在另一个例子中，医疗分析系统使用步进电机确定要分析的样本传送带的位置。结果发现将传送带回送到开始单元格的结构，常常造成机械手错过第一个单元格。

作为逻辑（相对于物理）变量的一个例子，我们可以研究PIN或电话号码。很难想像0000、0001、5000、9998和9999的PIN会发现什么缺陷。

5.2 健壮性测试

健壮性测试是边界值分析的一种简单扩展：除了变量的五个边界值分析取值，还要通过采用一个略超过最大值（max+）的取值，以及一个略小于最小值（min-）的取值，看看超过极值时系统会有什么表现。我们系列例子的健壮性测试用例如图5-3所示。

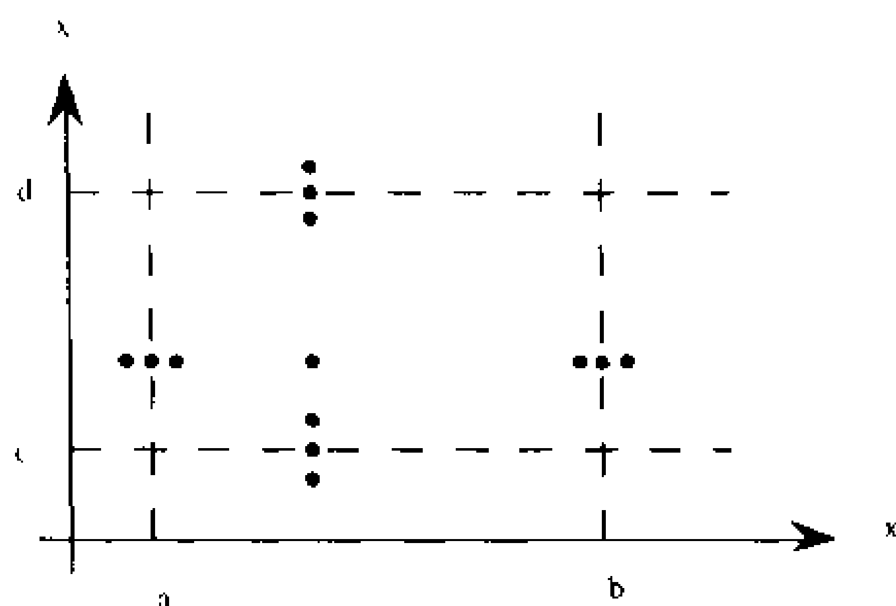


图5-3 两变量函数的健壮性测试用例

边界值分析的大部分讨论都直接适用于健壮性测试，尤其是归纳和局限性的讨论。健壮性测试最有意思的部分不是输入，而是预期的输出。当物理量超过其最大值时会出现什么情况呢？如果是飞机机翼的迎角，则飞机可能失速；如果是公共电梯的负荷能力，则我们不希望出现特别的情况。健壮性测试的主要价值是观察例外处理情况。对于强类型语言，健壮性测试可能非常困难。例如在Pascal中，如果变量被定义在特定范围内，则超过这个范围的取值都会产生导致中断正常执行的运行时错误。这带来实现理念的一个有意思的问题：最好是执行显式的范围检查并使用例外处理机制解决"健壮值"问题呢，还是坚持通过强类型来解决？如果采用例外处理选择，则必须进行健壮性测试。

5.3 最坏情况测试

正如我们在前面已经提到过的，边界值测试分析采用了可靠性理论的单缺陷假设。拒绝这种假设，意味着我们关心当多个变量取极值时会出现什么情况。在电子电路分析中，这叫做"最坏情况测试"。我们在这里使用这种思想来生成最坏情况测试用例。对每个变量，首先进行包含最小值、略高于最小值、正常值、略低于最大值和最大值五元素集合的测试，然后对这些集合进行笛卡儿积（请参阅第3章）计算，以生成测试用例。这种测试的两变量函数测试用例如图5-4所示。

最坏情况测试显然更彻底，因为边界值分析测试用例是最坏情况测试用例的真子集。最坏情况测试还意味着更多的工作量：n变量函数的最坏情况测试，会产生 5^n 个测试用例，而

边界值分析只产生4n + 1个测试用例。

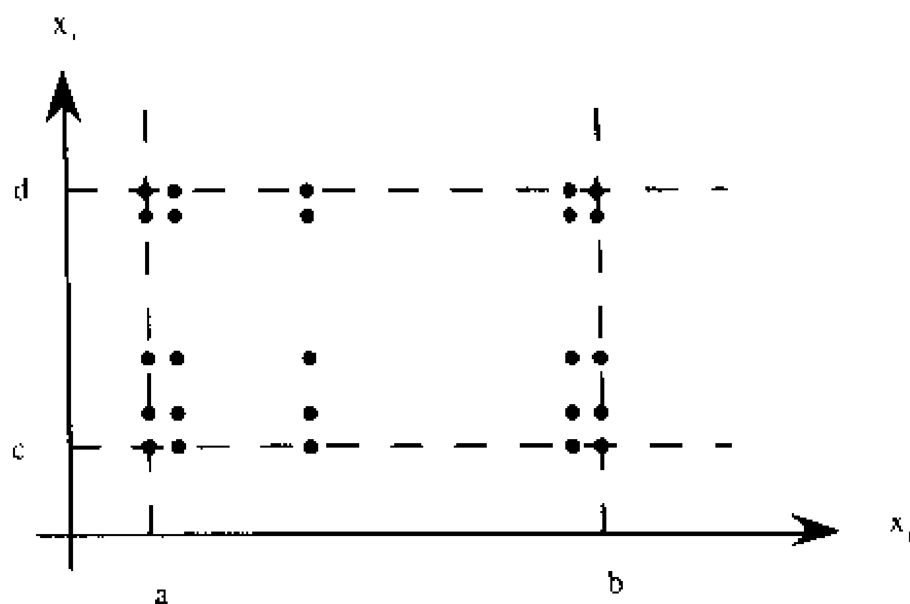


图5-4　两变量函数的最坏情况测试用例

最坏情况测试的归纳模式与边界值分析的归纳模式一样，两者也有相同的局限性，特别是独立性要求方面的局限性。最坏情况测试的最佳运用，可能是物理变量具有大量交互作用，或者函数失效的代价极高的情况。对于确实极端的测试，会采用健壮最坏情况测试，这种测试使用健壮性测试的七元素集合的笛卡儿积。图5-5给出了两变量函数的健壮最坏情况测试用例。

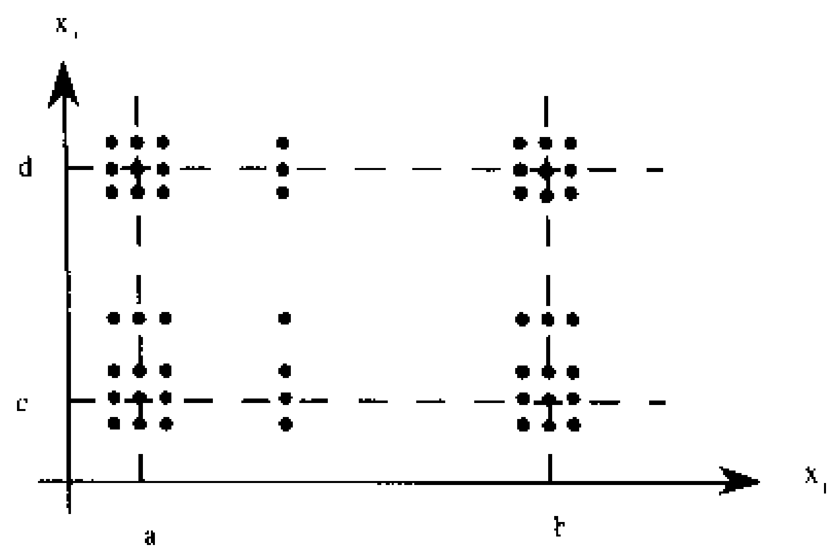


图5-5　两变量函数的健壮最坏情况测试用例

5.4　特殊值测试

特殊值测试大概是运用得最广泛的一种功能性测试，特殊值测试还最直观、最不一致。当测试人员使用其领域知识、使用类似程序的经验以及关于"软点"信息开发测试用例时，会出现特殊值测试。我们也可以把这种测试叫做"特别测试"和"人工"seat-of-the-pants或-skirt测试，不使用测试方针，只使用"最佳工程判断。"结果，特殊值测试特别依赖测试人员的能力。

尽管有所有这些缺点，特殊值测试还是非常有用的。在下一节中读者将会看到，通过我们刚刚讨论过的方法为三个例子生成的测试用例（ATM系统没有采用这种方法）。如果仔细研究这些例子，特别是NextDate函数，就会发现没有特别令人满意的地方。如果感兴趣的测试人员为NextDate定义特殊值测试用例，我们会发现多个测试用例会涉及2月28日、2月29日和闰年。尽管特殊值测试是高度主观性的，但是所产生的测试用例集合，常常比用我们已经研究过的其他方法生成的测试集合，更能有效地发现缺陷，这也说明了软件测试的工艺性质。

5.5 举例

这三个系列例子都是有三个变量的函数。列出每个问题通过所有方法生成的测试用例会很占篇幅，因此只挑出一些例子。

5.5.1 三角形问题的测试用例

在描述问题时，关于三角形的边，除了说明是整数外，没有给出其他条件。显然，低端边界都是1。我们任意取200作为高端边界。表5-1给出了使用这些值域产生的边界值测试用例，表5-2给出的是最坏情况测试用例。

表5-1 边界值分析测试用例

用例	a	b	c	预期输出
1	100	100	1	等腰三角形
2	100	100	2	等腰三角形
3	100	100	100	等边三角形
4	100	100	199	等腰三角形
5	100	100	200	非三角形
6	100	1	100	等腰三角形
7	100	2	100	等腰三角形
8	100	100	100	等边三角形
9	100	199	100	等腰三角形
10	100	200	100	非三角形
11	1	100	100	等腰三角形
12	2	100	100	等腰三角形
13	100	100	100	等边三角形
14	199	100	100	等腰三角形
15	200	100	100	非三角形

表5-2　最坏情况测试用例

用例	a	b	c	预期输出
1	1	1	1	等边三角形
2	1	1	2	非三角形
3	1	1	100	非三角形
4	1	1	199	非三角形
5	1	1	200	非三角形
6	1	2	1	非三角形
7	1	2	2	等腰三角形
8	1	2	100	非三角形
9	1	2	199	非三角形
10	1	2	200	非三角形
11	1	100	1	非三角形
12	1	100	2	非三角形
13	1	100	100	等腰三角形
14	1	100	199	非三角形
15	1	100	200	非三角形
16	1	199	1	非三角形
17	1	199	2	非三角形
18	1	199	100	非三角形
19	1	199	199	等腰三角形
20	1	199	200	非三角形
21	1	200	1	非三角形
22	1	200	2	非三角形
23	1	200	100	非三角形
24	1	200	199	非三角形
25	1	200	200	等腰三角形
26	2	1	1	非三角形
27	2	1	2	等腰三角形
28	2	1	100	非三角形
29	2	1	199	非三角形
30	2	1	200	非三角形
31	2	2	1	等腰三角形
32	2	2	2	等边三角形
33	2	2	100	非三角形
34	2	2	199	非三角形
35	2	2	200	非三角形
36	2	100	1	非三角形
37	2	100	2	非三角形
38	2	100	100	等腰三角形
39	2	100	199	非三角形
40	2	100	200	非三角形

（续）

用例	a	b	c	预期输出
41	2	199	1	非三角形
42	2	199	2	非三角形
43	2	199	100	非三角形
44	2	199	199	等腰三角形
45	2	199	200	不等边三角形
46	2	200	1	非三角形
47	2	200	2	非三角形
48	2	200	100	非三角形
49	2	200	199	不等边三角形
50	2	200	200	等腰三角形
51	100	1	1	非三角形
52	100	1	2	非三角形
53	100	1	100	等腰三角形
54	100	1	199	非三角形
55	100	1	200	非三角形
56	100	2	1	非三角形
57	100	2	2	非三角形
58	100	2	100	等腰三角形
59	100	2	199	非三角形
60	100	2	200	非三角形
61	100	100	1	等腰三角形
62	100	100	2	等腰三角形
63	100	100	100	等边三角形
64	100	100	199	等腰三角形
65	100	100	200	非三角形
66	100	199	1	非三角形
67	100	199	2	非三角形
68	100	199	100	等腰三角形
69	100	199	199	等腰三角形
70	100	199	200	不等边三角形
71	100	200	1	非三角形
72	100	200	2	非三角形
73	100	200	100	非三角形
74	100	200	199	不等边三角形
75	100	200	200	等腰三角形
76	199	1	1	非三角形
77	199	1	2	非三角形
78	199	1	100	非三角形
79	199	1	199	等腰三角形
80	199	1	200	非三角形

（续）

用例	a	b	c	预期输出
81	199	2	1	非三角形
82	199	2	2	非三角形
83	199	2	100	非三角形
84	199	2	199	等腰三角形
85	199	2	200	不等边三角形
86	199	100	1	非三角形
87	199	100	2	非三角形
88	199	100	100	等腰三角形
89	199	100	199	等腰三角形
90	199	100	200	不等边三角形
91	199	199	1	等腰三角形
92	199	199	2	等腰三角形
93	199	199	100	等腰三角形
94	199	199	199	等边三角形
95	199	199	200	等腰三角形
96	199	200	1	非三角形
97	199	200	2	不等边三角形
98	199	200	100	不等边三角形
99	199	200	199	等腰三角形
100	199	200	200	等腰三角形
101	200	1	1	非三角形
102	200	1	2	非三角形
103	200	1	100	非三角形
104	200	1	199	非三角形
105	200	1	200	等腰三角形
106	200	2	1	非三角形
107	200	2	2	非三角形
108	200	2	100	非三角形
109	200	2	199	不等边三角形
110	200	2	200	等腰三角形
111	200	100	1	非三角形
112	200	100	2	非三角形
113	200	100	100	非三角形
114	200	100	199	不等边三角形
115	200	100	200	等腰三角形
116	200	199	1	非三角形
117	200	199	2	不等边三角形
118	200	199	100	不等边三角形
119	200	199	199	等腰三角形
120	200	199	200	等腰三角形

（续）

用例	a	b	c	预期输出
121	200	200	1	等腰三角形
122	200	200	2	等腰三角形
123	200	200	100	等腰三角形
124	200	200	199	等腰三角形
125	200	200	200	等边三角形

5.5.2 NextDate函数的测试用例

表5-3 最坏情况测试用例

用例	月份	日期	年	预期输出
1	1	1	1812	1812年1月2日
2	1	1	1813	1813年1月2日
3	1	1	1912	1912年1月2日
4	1	1	2011	2011年1月2日
5	1	1	2012	2012年1月2日
6	1	2	1812	1812年1月3日
7	1	2	1813	1813年1月3日
8	1	2	1912	1912年1月3日
9	1	2	2011	2011年1月3日
10	1	2	2012	2012年1月3日
11	1	15	1812	1812年1月16日
12	1	15	1813	1813年1月16日
13	1	15	1912	1912年1月16日
14	1	15	2011	2011年1月16日
15	1	15	2012	2012年1月16日
16	1	30	1812	1812年1月31日
17	1	30	1813	1813年1月31日
18	1	30	1912	1912年1月31日
19	1	30	2011	2011年1月31日
20	1	30	2012	2012年1月31日
21	1	31	1812	1812年2月1日
22	1	31	1813	1813年2月1日
23	1	31	1912	1912年2月1日
24	1	31	2011	2011年2月1日
25	1	31	2012	2012年2月1日
26	2	1	1812	1812年2月2日
27	2	1	1813	1813年2月2日
28	2	1	1912	1912年2月2日
29	2	1	2011	2011年2月2日
30	2	1	2012	2012年2月2日

（续）

用例	月份	日期	年	预期输出
31	2	2	1812	1812年2月3日
32	2	2	1813	1813年2月3日
33	2	2	1912	1912年2月3日
34	2	2	2011	2011年2月3日
35	2	2	2012	2012年2月3日
36	2	15	1812	1812年2月16日
37	2	15	1813	1813年2月16日
38	2	15	1912	1912年2月16日
39	2	15	2011	2011年2月16日
40	2	15	2012	2012年2月16日
41	2	30	1812	错误
42	2	30	1813	错误
43	2	30	1912	错误
44	2	30	2011	错误
45	2	30	2012	错误
46	2	31	1812	错误
47	2	31	1813	错误
48	2	31	1912	错误
49	2	31	2011	错误
50	2	31	2012	错误
51	6	1	1812	1812年6月1日
52	6	1	1813	1813年6月1日
53	6	1	1912	1912年6月2日
54	6	1	2011	2011年6月2日
55	6	1	2012	2012年6月2日
56	6	2	1812	1812年6月3日
57	6	2	1813	1813年6月3日
58	6	2	1912	1912年6月3日
59	6	2	2011	2011年6月3日
60	6	2	2012	2012年6月3日
61	6	15	1812	1812年6月16日
62	6	15	1813	1813年6月16日
63	6	15	1912	1912年6月16日
64	6	15	2011	2011年6月16日
65	6	15	2012	2012年6月16日
66	6	30	1812	1812年7月31日
67	6	30	1813	1813年7月31日
68	6	30	1912	1912年7月31日
69	6	30	2011	2011年7月31日
70	6	30	2012	2012年7月31日

（续）

用例	月份	日期	年	预期输出
71	6	31	1812	错误
72	6	31	1813	错误
73	6	31	1912	错误
74	6	31	2011	错误
75	6	31	2012	错误
76	11	1	1812	1812年11月2日
77	11	1	1813	1813年11月2日
78	11	1	1912	1912年11月2日
79	11	1	2011	2011年11月2日
80	11	1	2012	2012年11月2日
81	11	2	1812	1812年11月3日
82	11	2	1813	1813年11月3日
83	11	2	1912	1912年11月3日
84	11	2	2011	2011年11月3日
85	11	2	2012	2012年11月3日
86	11	15	1812	1812年11月16日
87	11	15	1813	1813年11月16日
88	11	15	1912	1912年11月16日
89	11	15	2011	2011年11月16日
90	11	15	2012	2012年11月16日
91	11	30	1812	1812年12月1日
92	11	30	1813	1813年12月1日
93	11	30	1912	1912年12月1日
94	11	30	2011	2011年12月1日
95	11	30	2012	2012年12月1日
96	11	31	1812	错误
97	11	31	1813	错误
98	11	31	1912	错误
99	11	31	2011	错误
100	11	31	2012	错误
101	12	1	1812	1812年12月2日
102	12	1	1813	1813年12月2日
103	12	1	1912	1912年12月2日
104	12	1	2011	2011年12月2日
105	12	1	2012	2012年12月2日
106	12	2	1812	1812年12月3日
107	12	2	1813	1813年12月3日
108	12	2	1912	1912年12月3日
109	12	2	2011	2011年12月3日
110	12	2	2012	2012年12月3日

（续）

用例	月份	日期	年	预期输出
111	12	15	1812	1812年12月16日
112	12	15	1813	1813年12月16日
113	12	15	1912	1912年12月16日
114	12	15	2011	2011年12月16日
115	12	15	2012	2012年12月16日
116	12	30	1812	1812年12月31日
117	12	30	1813	1813年12月31日
118	12	30	1912	1912年12月31日
119	12	30	2011	2011年12月31日
120	12	30	2012	2012年12月31日
121	12	31	1812	1813年1月1日
122	12	31	1813	1814年1月1日
123	12	31	1912	1913年1月1日
124	12	31	2011	2012年1月1日
125	12	31	2012	2013年1月1日

5.5.3　佣金问题的测试用例

我们不再列出125个乏味的测试用例，而是研究佣金问题的一些更有意思的测试用例。这一次我们将研究输出值域的边界值，特别是接近1000美元和1800美元门限点的值。佣金的输出空间如图5-6所示。图中给出了带有坐标轴的门限平面的截面。

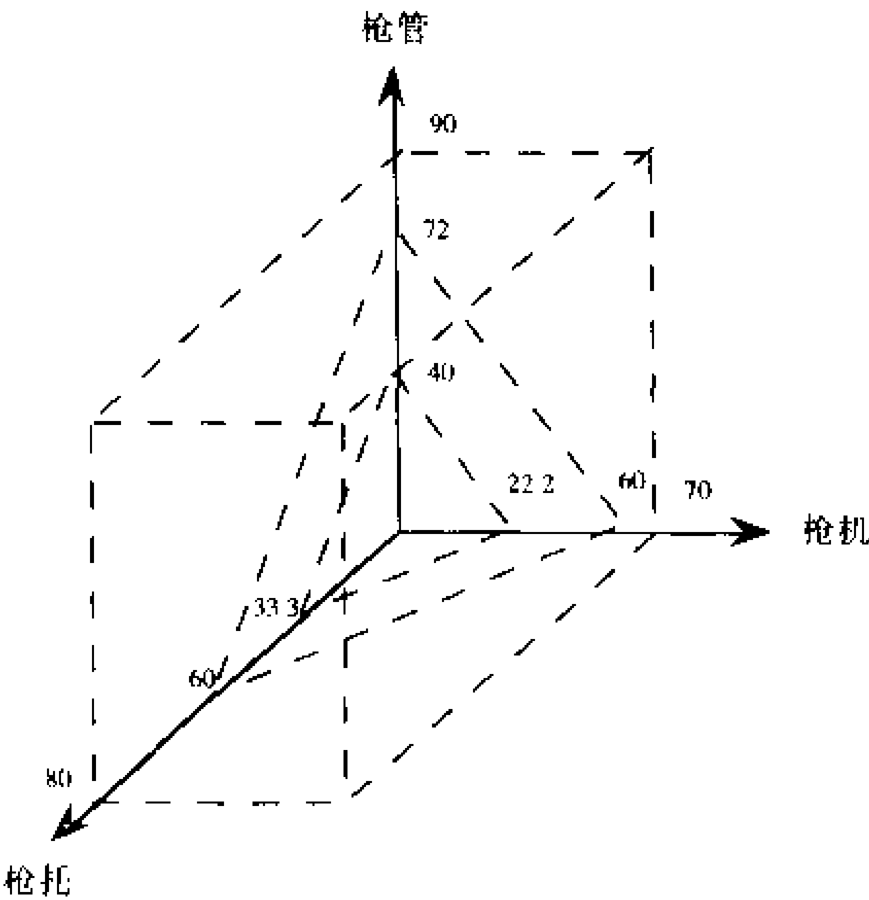


图5-6　佣金问题的输入空间

表5-4 输出边界值分析测试用例

用例	枪机	枪托	枪管	销售额	佣金	注释
1	1	1	1	100	10	输出最小值
2	1	1	2	125	12.5	输出略大于最小值
3	1	2	1	130	13	输出略大于最小值
4	2	1	1	145	14.5	输出略大于最小值
5	5	5	5	500	50	中点
6	10	10	9	975	97.5	略低于边界点
7	10	9	10	970	97	略低于边界点
8	9	10	10	955	95.5	略低于边界点
9	10	10	10	1000	100	边界点
10	10	10	11	1025	103.75	略高于边界点
11	10	11	10	1030	104.5	略高于边界点
12	11	10	10	1045	106.75	略高于边界点
13	14	14	14	1400	160	中点
14	18	18	17	1775	216.25	略低于边界点
15	18	17	18	1770	215.5	略低于边界点
16	17	18	18	1755	213.25	略低于边界点
17	18	18	18	1800	220	边界点
18	18	18	19	1825	225	略高于边界点
19	18	19	18	1830	226	略高于边界点
20	19	18	18	1845	229	略高于边界点
21	48	48	48	4800	820	中点
22	70	80	89	7775	1415	输出略小于最大值
23	70	79	90	7770	1414	输出略小于最大值
24	69	80	90	7755	1411	输出略小于最大值
25	70	80	90	7800	1420	输出最大值

低于较低平面的值，对应低于1000美元门限的销售额。两个平面之间的值是15%佣金区域。使用输出值域确定测试用例的部分原因是，通过输入值域生成的测试用例几乎都在20%区域。我们要找出强调边界值100美元、1000美元、1800美元和7800美元对应的输入变量组合。这些测试用例是通过电子表格开发的，节省了大量计算工作。最大值和最小值的确定很容易，给出的数正好便于生成边界点。有意思的是：测试用例9是1000美元的边界点。如果调整输入变量，则可以得到略低和略高于该边界的值（测试用例6~8和10~12）。如果愿意，还可以选择接近截面的值，例如（22，1，1）和（21，1，1）。如果继续采用这种方式测试，那么就是在“运行”代码的重要部分。我们可以说这确实是一种特殊值测试，因为我们利用了数学知识来生成测试用例。

表5-5 输出特殊值测试用例

用例	枪机	枪托	枪管	销售额	佣金	注释
1	10	11	9	1005	100.75	略高于边界点
2	18	17	19	1795	219.25	略低于边界点
3	18	19	17	1805	221	略高于边界点

5.6 随机测试

已出版的文献讨论随机测试问题已经至少有20年的历史了。主要是学术界在统计学方面对这个问题感兴趣。我们的三个示例问题很适合随机测试。随机测试的基本思想是，不是永远选取有界变量的最小值、略高于最小值、正常值、略低于最大值和最大值，而是使用随机数生成器选出测试用例值。随机测试可以避免出现测试偏见，但是也带来一个严重问题：多少随机测试用例才是充分的？稍后我们在讨论结构性测试覆盖率指标时，会得到一种很好的答案。表5-6、5-7和5-8给出了随机生成的测试用例结果，这些测试用例都是用选取有界变量 $a \leq x \leq b$ 值的一个Visual Basic应用程序生成的，x满足下式：

$$x = \mathrm{Int}\,(\, b - a + 1\,) * \mathrm{Rnd} + a$$

其中函数Int返回浮点数的整数部分，函数Rnd生成区间[0, 1]内的随机数。这个程序持续生成随机测试用例，直到每种输出至少出现一次。在每张表中，该程序要进行七次“循环”，以“很难生成”的测试用例结束。

表5-6　三角形程序的随机测试用例

测试用例	非三角形	不等边三角形	等腰三角形	等边三角形
1289	663	593	32	1
15436	7696	7372	367	1
17091	8556	8164	367	1
2603	1284	1252	66	1
6475	3197	3122	155	1
5978	2998	2850	129	1
9008	4447	4353	207	1
平均值	49.83%	47.87%	2.29%	0.01%

表5-7　佣金程序的随机测试用例

测试用例	10%	15%	20%
91	1	6	84
27	1	1	25
72	1	1	70
176	1	6	169
48	1	1	46
152	1	6	145
125	1	4	120
平均值	1.01%	3.62%	95.37%

表5-8　NextDate程序的随机测试用例

测试用例	有31天的月份的1~30日	有31天的月份的31日	有30天的月份的1~29日	有30天的月份的30日
913	542	17	274	10
1101	621	9	358	8
4201	2448	64	1242	46
1097	600	21	350	9
5853	3342	100	1804	82
3959	2195	73	1252	42
1436	786	22	456	13
平均值	56.76%	1.65%	30.91%	1.13%
可能值	56.45%	1.88%	31.18%	1.88%
2月的1~27日	闰年的2月28日	非闰年的2月28日	闰年的2月29日	不可能的日期
45	1	1	1	22
83	1	1	1	19
312	1	8	3	77
92	1	4	1	19
417	1	11	2	94
310	1	6	5	75
126	1	5	1	26
7.46%	0.04%	0.19%	0.08%	1.79%
7.26%	0.07%	0.20%	0.07%	1.01%

5.7 边界值测试的指导方针

除了特殊值测试，基于函数（程序）输入定义域的测试方法，是所有测试方法中最基本的。这类测试方法都有一种假设，即输入变量是真正独立的，如果不能保证这种假设，则这类方法会产生不能令人满意的测试用例（例如在NextDate中生成1912年2月31日）。这些方法还有两方面的区别：正常值与健壮值，单缺陷与多缺陷假设。仅仅仔细地运用这些差别就能够产生较好的测试。这些方法都可以用于程序的输出值域，就像我们在佣金问题中所做的那样。

另一种很有用的基于输出的测试用例形式，可用于生成错误消息的系统。测试人员应该设计测试用例，以检查在适当的时候，错误消息是否被生成，并且不会被错误地生成。定义域分析还可以用于内部变量，例如循环控制变量、索引和指针。严格地说，这些都不是输入变量，但是这些变量的使用错误相当常见。健壮性测试是测试内部变量的一种好的选择。

5.8 练习

1. 请导出有n个变量的函数的健壮性测试用例个数的公式。

2. 请导出有n个变量的函数的健壮最坏情况测试用例个数的公式

3. 请画出维恩图来表示边界值分析、健壮性测试、最坏情况测试和健壮性最坏情况测试生成的测试用例之间的关系

4. 如果试图进行输出值域健壮性测试，会出现什么情况？请以佣金问题为例说明。

5. 如果读者做了第2章的练习8，就会熟悉CRC出版公司网站（http://www.crcpress.com）的下载过程。通过这个网站，可以下载叫做BlackBox.exe的可执行Visual Basic程序。（它有一个Naive.exe面向文件的版本，也包含同样的插入缺陷。）文件A1.txt、B1.txt和C1.txt分别包含三角形、NextDate和佣金问题的最坏情况边界值测试用例。请运行这些测试用例集合，在读者自己命名的文件中保存测试结果，并与第2章自然测试结果进行比较。

第6章 等价类测试

使用等价类作为功能性测试的基础有两个动机：我们希望进行完备的测试，同时又希望避免冗余。边界值测试不能实现这两种希望中的任意一个：研究那些测试用例表，很容易看出存在大量冗余，再进一步仔细研究，还会发现严重漏洞。等价类测试重复边界值测试的两个决定因素，即健壮性和单/多缺陷假设。本书的第1版给出了三种形式的等价类测试，这里我们再给出第四种。单缺陷与多缺陷假设产生第1版的弱/强等价类测试之分。针对是否进行无效数据的处理又产生健壮与一般等价类测试之分。

大多数标准测试教科书（Myers，1979；Mosley，1993）都讨论了我们叫做弱健壮等价类测试的问题。这种传统形式的测试关注无效数据值，体现了20世纪60年代和70年代主流程序设计风格。输入数据检验在当时是一个很重要的问题，“垃圾入，垃圾出”曾经是程序员的格言。对这种问题的通常响应是在程序中加入大量输入检验代码。很多文献的作者和研讨会主持人常常提到，在经典的输入/中心处理/输出的结构化程序体系结构中，输入部分常常占总源程序的80%。在这种情况下，强调输入数据的检验是很自然的。随着逐渐向现代程序设计语言的转移，特别是向那些具有强数据类型功能的语言，然后又向图形用户界面（GUI）语言的转移，使得输入数据检验不再那么重要。

6.1 等价类

第3章曾经指出，等价类的重要问题是它们构成集合的划分，其中，划分是指互不相交的一组子集，这些子集的并是整个集合。这对于测试有两点非常重要的意义：表示整个集合这个事实提供了一种形式的完备性，而互不相交可保证一种形式的无冗余性。由于子集是由等价关系决定的，因此子集的元素都有一些共同点。等价类测试的思想是通过每个等价类中的一个元素标识测试用例。如果广泛选择等价类，则这样可以大大降低测试用例之间的冗余。例如，在三角形问题中，我们当然要有一个等边三角形的测试用例，我们可能选择三元组（5，5，5）作为测试用例的输入。如果这样做了，则可预期不会从诸如（6，6，6）和（100，100，100）这样的测试用例中得到多少新东西。直觉告诉我们，这些测试用例会以与第一个测试用例一样的方式进行“相同处理”，因此，这些测试用例是冗余的。当我们在第三部分考虑结构性测试时，将会看到“相同处理”映射到“遍历相同的执行路径”。

等价类测试的关键（以及工艺！），就是选择确定类的等价关系。我们常常通过预测可能的实现，并考虑在实现中必须提供的功能操作来做出这种选择。我们将用系列例子说明这一点，但是首先必须区分弱和强等价类测试。然后，我们将这些测试与传统形式的等价类

测试进行比较。

需要丰富在边界值测试所使用的函数。同样为了便于理解，我们还是将讨论与一个两变量 x_1 和 x_2 函数F联系起来。如果F实现为一个程序，则输入变量 x_1 和 x_2 将拥有以下边界，以及边界内的区间：

$a \leq x_1 \leq d$，区间为[a，b)，[b，c)，[c，d]

$e \leq x_2 \leq g$，区间为[e，f)、[f，g]

其中方括号和圆括号分别表示闭区间和开区间的端点。x_1 和 x_2 的无效值是：$x_1<a$，$x_1>d$，以及$x_2<e$，$x_2>g$。

6.1.1 弱一般等价类测试

采用前面给出的标记，弱一般等价类测试通过使用一个测试用例中的每个等价类（区间）的一个变量实现。（请注意单缺陷假设的作用。）对于前面给出的例子，可得到如图6-1所示的弱等价类测试用例。

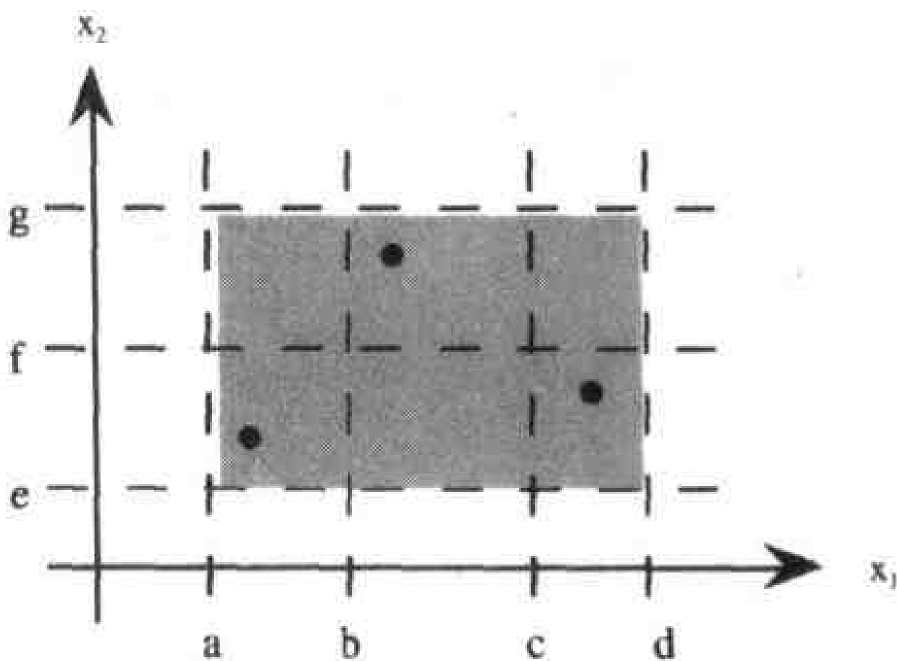


图6-1 弱一般等价类测试用例

这三个测试用例使用每个等价类中的一个值。我们以对称方式标识这些测试用例，于是得到外在的模式。事实上，永远都有等量的弱等价类测试用例，因为划分中的类对应最大子集数。

6.1.2 强一般等价类测试

强一般等价类测试基于多缺陷假设，因此需要等价类笛卡儿积的每个元素对应的测试用例，如图6-2所示。

请注意，这些测试用例的模式与命题逻辑中的真值表构造具有相似性。笛卡儿积可保证两种意义上的“完备性”：一是覆盖所有的等价类，二是有可能的输入组合中的一个。

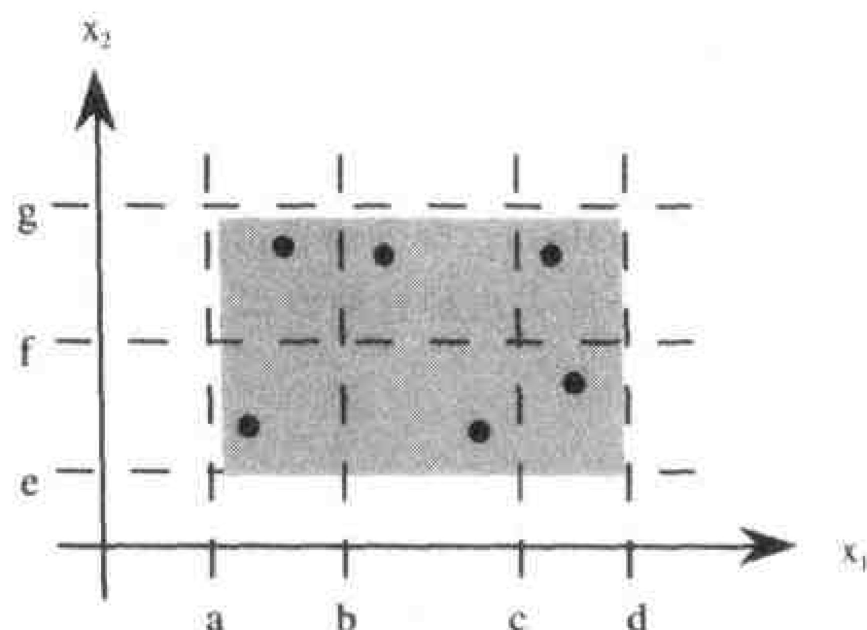


图6-2 强一般等价类测试用例

我们将会通过系列例子看到，“好的”等价类测试的关键是等价关系的选择。注意被“相同处理”的输入。在大多数情况下，等价类测试定义输入定义域的类。没有理由不能根据被测程序函数的输出值域定义等价关系。事实上，这对于三角形问题是最简单的方法。

6.1.3 弱健壮等价类测试

这种测试的名称显然与直觉矛盾，且自相矛盾。怎么能够既弱又健壮呢？说它健壮，是因为这种测试考虑了无效值；说它弱，是因为有单缺陷假设。（本书第1版将这种测试叫做“传统等价类测试”。）

1. 对于有效输入，使用每个有效类的一个值（就像我们在所谓弱一般等价类测试中所做的一样。请注意，这些测试用例中的所有输入都是有效的。）

2. 对于无效输入，测试用例将拥有一个无效值，并保持其余的值都是有效的。（因此，“单缺陷”会造成测试用例失败。）

按照这种策略产生的测试用例如图6-3所示。

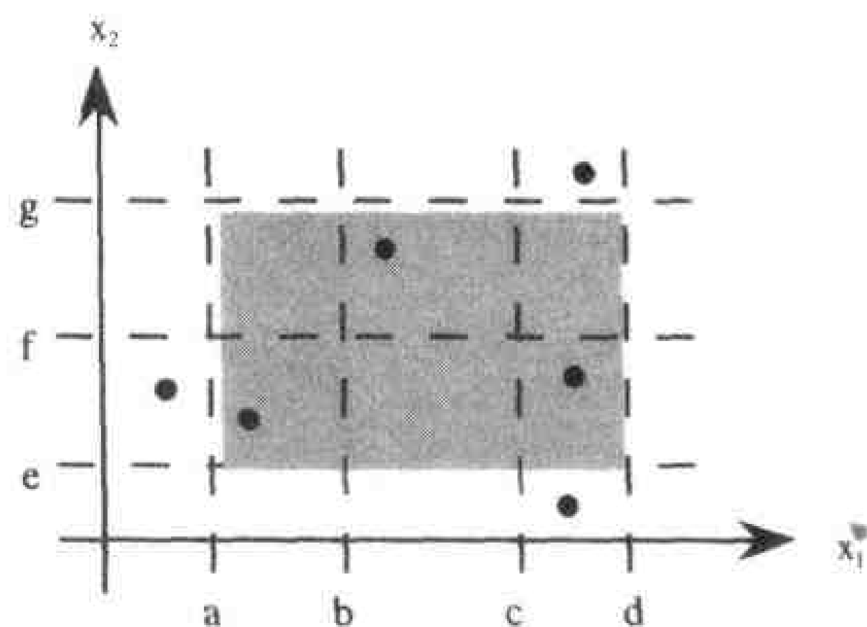


图6-3 弱健壮等价类测试用例

健壮等价类测试有两个问题。第一个问题是，规格说明常常并没有定义无效测试用例所预期的输出是什么。（我们可以把这看做是规格说明的不足，但是这并不能解决问题。）因此，测试人员需要花费大量时间定义这些测试用例的输出。第二个问题是，强类型语言没有必要考虑无效输入。传统等价类测试是诸如FORTRAN和COBOL这样的语言占统治地位的年代的产物，因此那时这种错误很常见。事实上，正是由于经常出现这种错误，才促使人们实现强类型语言。

6.1.4 强健壮等价类测试

至少这种测试的名称既不与直觉矛盾，也不自相矛盾，只是有些冗余。像以前的定义一样，健壮是指要考虑无效值，强是指多缺陷假设。（本书第1版没有讨论这种形式的测试）。

我们从所有等价类笛卡儿积的每个元素中获得测试用例，如图6-4所示。

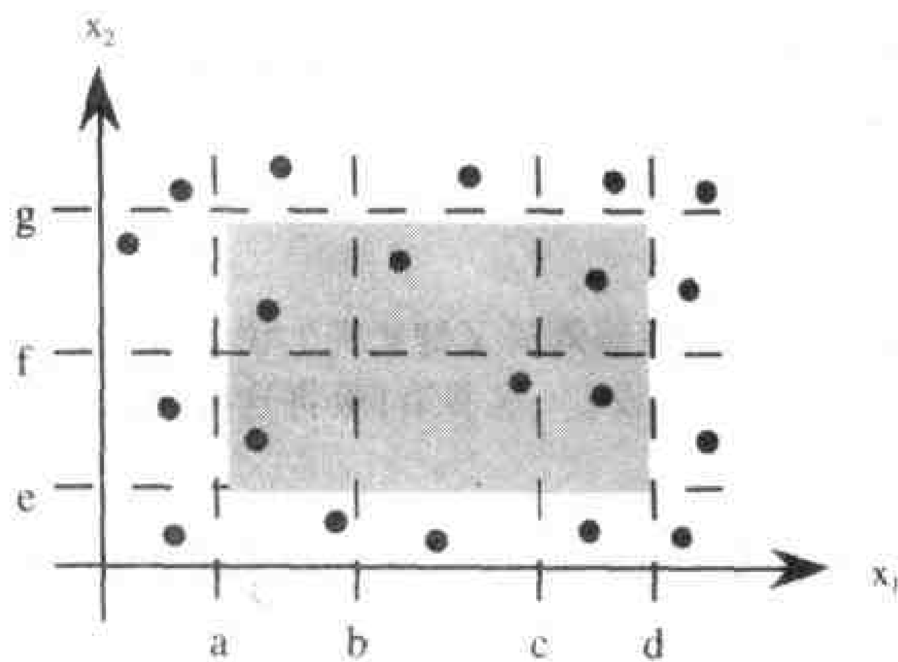


图6-4 强健壮等价类测试用例

6.2 三角形问题的等价类测试用例

在描述问题时，我们曾经提到有四种可能出现的输出：非三角形、不等边三角形、等腰三角形和等边三角形。可以使用这些输出标识如下所示的输出（值域）等价类：

R1 = {<a, b, c>: 有三条边a、b和c的等边三角形}
R2 = {<a, b, c>: 有三条边a、b和c的等腰三角形}
R3 = {<a, b, c>: 有三条边a、b和c的不等边三角形}
R4 = {<a, b, c>: 三条边a、b和c不构成三角形}

四个弱一般等价类测试用例是：

测试用例	a	b	c	预期输出
WN1	5	5	5	等边三角形
WN2	2	2	3	等腰三角形
WN3	3	4	5	不等边三角形
WN4	4	1	2	非三角形

由于变量a、b和c没有有效区间，则强一般等价类测试用例与弱一般等价类测试用例相同。

考虑a、b和c的无效值产生的以下额外弱健壮等价类测试用例：

测试用例	a	b	c	预期输出
WR1	-1	5	5	a取值不在所允许的取值值域内
WR2	5	-1	5	b取值不在所允许的取值值域内
WR3	5	5	-1	c取值不在所允许的取值值域内
WR4	201	5	5	a取值不在所允许的取值值域内
WR5	5	201	5	b取值不在所允许的取值值域内
WR6	5	5	201	c取值不在所允许的取值值域内

以下是额外强健壮等价类测试用例三维立方的一个“角”：

测试用例	a	b	c	预期输出
SR1	-1	5	5	a取值不在所允许的取值值域内
SR2	5	-1	5	b取值不在所允许的取值值域内
SR3	5	5	-1	c取值不在所允许的取值值域内
SR4	-1	-1	5	a、b取值不在所允许的取值值域内
SR5	5	-1	-1	b、c取值不在所允许的取值值域内
SR6	-1	5	-1	a、c取值不在所允许的取值值域内
SR7	-1	-1	-1	a、b、c取值不在所允许的取值值域内

请注意，预期输出如何完备地描述无效输入值。

等价类测试显然对用来定义类的等价关系很敏感。这是工艺特性的又一个例子。如果在输入定义域上定义等价类，则可以得到更丰富的测试用例集合。三个整数a、b和c有些什么可能的取值呢？这些整数可以都相等，有一对整数相等（有三种相等方式），或都不相等：

$D1 = \{<a, b, c>: a = b = c\}$

$D2 = \{<a, b, c>: a = b, a \neq c\}$

$D3 = \{<a, b, c>: a = c, a \neq b\}$

$D4 = \{<a, b, c>: b = c, a \neq b\}$

$D5 = \{<a, b, c>: a \neq b, a \neq c, b \neq c\}$

作为一个单独的问题，我们可以通过三角形的性质来判断三条边是否构成一个三角形。（例如，三元组<1，4，1>有一对相等的边，但是这些边不构成一个三角形。）

D6 = {<a，b，c>：a≥b + c}

D7 = {<a，b，c>：b≥a + c}

D8 = {<a，b，c>：c≥a + b}

如果我们要彻底一些，可以将“小于或等于”分解为两种不同的情况，这样D6就变成：

D6' = {<a，b，c>：a = b + c}

D6" = {<a，b，c>：a > b + c}

同样对于D7和D8也有类似的情况。

6.3 NextDate函数的等价类测试用例

NextDate函数可以很好地说明选择内部等价关系的工艺。前面已经介绍过，NextDate是一个三变量函数，即月份、日期和年，这些变量的有效值区间定义如下：

M1 = {月份：1≤月份≤12}

D1 = {日期：1≤日期≤31}

Y1 = {年：1812≤年≤2012}

无效等价类是：

M2 = {月份：月份 < 1}

M3 = {月份：月份 > 12}

D2 = {日期：日期 < 1}

D3 = {日期：日期 > 31}

Y2 = {年：年 < 1812}

Y3 = {年：年 > 2012}

由于有效类的数量等于独立变量的个数，因此只有弱一般等价类测试用例出现，并且与强一般等价类测试用例相同：

用例ID	月份	日期	年	预期输出
WN1，SN1	6	15	1912	1912年6月16日

以下是弱健壮测试用例的完整集合：

用例ID	月份	日期	年	预期输出
WR1	6	15	1912	1912年6月16日
WR2	-1	15	1912	月份不在有效值域1..12中
WR3	13	15	1912	月份不在有效值域1..12中
WR4	6	-1	1912	日期不在有效值域1..31中
WR5	6	32	1912	日期不在有效值域1..31中
WR6	6	15	1811	年不在有效值域1812..2012中
WR7	6	15	2013	年不在有效值域1812..2012中

与三角形问题一样，以下是额外强健壮等价类测试用例三维立方的一个"角"：

用例ID	月份	日期	年	预期输出
SR1	-1	15	1912	月份不在有效值域1..12中
SR2	6	-1	1912	日期不在有效值域1..31中
SR3	6	15	1811	年不在有效值域1812..2012中
SR4	-1	-1	1912	月份不在有效值域1..12中 日期不在有效值域1..31中
SR5	6	-1	1811	日期不在有效值域1..31中 年不在有效值域1812..2012中
SR6	-1	15	1811	月份不在有效值域1..12中 年不在有效值域1812..2012中
SR7	-1	-1	1811	月份不在有效值域1..12中 日期不在有效值域1..31中 年不在有效值域1812..2012中

如果更仔细地选择等价关系，所得到的等价类将会更有用。前面曾经提到过，等价关系的要点是，类中的元素要被"同样处理"。理解传统方法不足的一种方法，是注意到"处理"在有效/无效层次上进行。通过关注更具体的处理可降低粒度。

必须对输入日期做怎样的处理？如果它不是某个月的最后一天，则NextDate函数会直接对日期加1。到了月末，下一个日期是1，月份加1。到了年末，日期和月份都会复位到1，年加1。最后，闰年问题要确定有关的月份的最后一天。经过这些分析之后，可以假设以下等价类：

M1 = {月份：每月有30天}

M2 = {月份：每月有31天}

M3 = {月份：此月是2月}

D1 = {日期：1≤日期≤28}

D2 = {日期：日期 = 29}

D3 = {日期：日期 = 30}

D4 = {日期：日期 = 31}

Y1 = {年：年 = 2000}

Y2 = {年：年是闰年}

Y3 = {年：年是平年}

通过选择有30天的月份和有31天的月份的独立类，可以简化月份最后一天问题。通过把2月分成独立的类，可以对闰年问题给予更多关注。我们还要特别关注日期的值：D1中的日（差不多）总是加1，D4中的日只对M2中的月才有意义。最后，年有三个类，包括2000年这个特例、闰年和非闰年类。这并不是完美的等价类集合，但是通过这种等价类集合可以发现很多潜在错误。

等价类测试用例

这些类产生以下弱等价类测试用例。与以前一样，机械地从对应类的近似中间选择输入：

用例ID	月份	日期	年	预期输出
WN1	6	14	2000	2000年6月15日
WN2	7	29	1996	1996年7月30日
WN3	2	30	2002	2002年2月31日（不可能的日期）
WN4	6	31	2000	2000年7月1日（不可能的输入日期）

机械选择输入值不考虑领域知识，因此没有考虑两种不可能出现的日期。"自动"测试用例生成永远都会有这种问题，因为领域知识不是通过等价类选择获得的。经过改进的强一般等价类测试用例是：

用例ID	月份	日期	年	预期输出
SN1	6	14	2000	2000年6月15日
SN2	6	14	1996	1996年6月15日
SN3	6	14	2002	2002年6月15日
SN4	6	29	2000	2000年6月30日
SN5	6	29	1996	1996年6月30日
SN6	6	29	2002	2002年6月30日
SN7	6	30	2000	2000年6月31日（不可能的日期）
SN8	6	30	1996	1996年6月31日（不可能的日期）
SN9	6	30	2002	2002年6月31日（不可能的日期）
SN10	6	31	2000	2000年7月1日（无效的输入日期）
SN11	6	31	1996	1996年7月1日（无效的输入日期）
SN12	6	31	2002	2002年7月1日（无效的输入日期）
SN13	7	14	2000	2000年7月15日
SN14	7	14	1996	1996年7月15日
SN15	7	14	2002	2002年7月15日
SN16	7	29	2000	2000年7月30日

（续）

用例ID	月份	日期	年	预期输出
SN17	7	29	1996	1996年7月30日
SN18	7	29	2002	2002年7月30日
SN19	7	30	2000	2000年7月31日
SN20	7	30	1996	1996年7月31日
SN21	7	30	2002	2002年7月31日
SN22	7	31	2000	2000年8月1日
SN23	7	31	1996	1996年8月1日
SN24	7	31	2002	2002年8月1日
SN25	2	14	2000	2000年2月15日
SN26	2	14	1996	1996年2月15日
SN27	2	14	2002	2002年2月15日
SN28	2	29	2000	2000年3月1日（无效的输入日期）
SN29	2	29	1996	1996年3月1日
SN30	2	29	2002	2002年3月1日（不可能的日期）
SN31	2	30	2000	2000年3月1日（不可能的日期）
SN32	2	30	1996	1996年3月1日（不可能的日期）
SN33	2	30	2002	2002年3月1日（不可能的日期）
SN34	6	31	2000	2000年7月1日（不可能的日期）
SN35	6	31	1996	1996年7月1日（不可能的日期）
SN36	6	31	2002	2002年7月1日（不可能的日期）

从弱一般测试转向强一般测试会产生一些边界值测试中也出现的冗余问题。从弱到强的转换，不管是一般类还是健壮类，都要做独立性假设，都要以等价类的叉积表示。3个月份类乘以4个日期类乘以3个年类，产生36个强一般等价类测试用例。对每个变量加上2个无效类，得到150个强健壮等价类测试用例（太多了，不能在这里给出）。

通过更仔细地研究年类，还可以精简测试用例集合。如果合并Y1和Y3，把结果称做平年，则36个测试用例就会降低到24个。这种变化不再特别关注2000年，并会增加判断闰年的难度。需要在难度和能够从当前测试用例中了解到的内容之间做平衡综合考虑。

6.4 佣金问题的等价类测试用例

佣金问题的输入定义域，由于枪机、枪托和枪管的限制而被“自然地”划分，这些等价类也正是通过传统等价类测试所标识的等价类。第一个类是有效输入，其他两个类是无效输入。输入定义域等价类产生非常不能令人满意的测试用例集合。对佣金函数的输出值域定义等价类可以改进测试用例集合。

输入变量有效类是：

L1 = {枪机：1≤枪机≤70}

L2 = {枪机 = -1}

S1 = {枪托：1≤枪托≤80}

B1 = {枪管：1≤枪管≤90}

输入变量对应的无效类是：

L3 = {枪机：枪机 = 0或枪机 < -1}

L3 = {枪机：枪机 > 70}

S2 = {枪托：枪托 < 1}

S3 = {枪托：枪托 > 80}

B2 = {枪管：枪管 < 1}

B3 = {枪管：枪管 > 90}

但是有一个问题。变量枪机还用做指示不再有电报的标记。当枪机等于 -1时，While循环就会终止，总枪机、总枪托和总枪管的值就会被用来计算销售额，进而计算佣金。

除了变量的名称和端点值区间不同之外，与NextDate函数的第一个版本完全相同。因此，也只有一个弱一般等价类测试用例，这个测试用例同样也等于强一般等价类测试用例。同样也有七个弱健壮测试用例。最后，额外强健壮等价类测试用例三维立方的一个"角"是：

用例ID	枪机	枪托	枪管	预期输出
SR1	-1	40	45	枪机值不在有效值域1..70中
SR2	35	-1	45	枪托值不在有效值域1..80中
SR3	35	40	-1	枪管值不在有效值域1..90中
SR4	-1	-1	45	枪机值不在有效值域1..70中 枪托值不在有效值域1..80中
SR5	-1	40	-1	枪机值不在有效值域1..70中 枪管值不在有效值域1..90中
SR6	35	-1	-1	枪托值不在有效值域1..80中 枪管值不在有效值域1..90中
SR7	-1	-1	-1	枪机值不在有效值域1..70中 枪托值不在有效值域1..80中 枪管值不在有效值域1..90中

6.4.1　输出值域等价类测试用例

请注意，对于强测试用例，不管是强一般测试用例还是强健壮测试用例，都只有一个是合理输入。如果确实担心错误案例，那么这就是很好的测试用例集合。但是很难确信佣金问题的计算部分没有问题。通过对输出值域定义等价类会有一定帮助。前面提到过，销售

额是所售出的枪机、枪托和枪管数量的函数：

销售额 = 45 × 枪机 + 30 × 枪托 + 25 × 枪管

我们可以根据佣金值域定义三个变量的等价类：

S1 = {<枪机，枪托，枪管>：销售额 ≤ 1000}

S2 = {<枪机，枪托，枪管>：1000 < 销售额 ≤ 1800}

S3 = {<枪机，枪托，枪管>：销售额 > 1800}

图5-6有助于更好地理解输入空间。S1的元素是接近原点金字塔中的整数点，S2的元素是金字塔与其他输入空间之间的“三角片”，S3的元素是不在S1和S2中的立方体中的点。输入定义域的强等价类所发现的错误案例都在图5-6所示立方体之外。

与三角形问题一样，输入是三元组这个事实意味着不再通过笛卡儿积获得测试用例。

6.4.2 输出值域等价类测试用例

测试用例	枪机	枪托	枪管	销售额	佣金
OR1	5	5	5	500	50
OR2	15	15	15	1500	175
OR3	25	25	25	2500	360

这些测试用例使我们有些感觉到正在接触问题的重要部分。与弱健壮测试用例结合在一起，可得到佣金问题的相当不错的测试，我们可能希望增加一些边界检查，只是为了保证从1000美元到1800美元的转移是正确的。这并不是特别容易，因为我们只能选择枪机、枪托和枪管。碰巧这个例子中的约束设计使三元组的数据“搭配合适”。

6.5 指导方针和观察

我们已经介绍了三个例子，最后讨论关于等价类测试的一些观察和等价类测试指导方针

1. 显然，等价类测试的弱形式（一般或健壮）不如对应的强形式的测试全面。

2. 如果实现语言是强类型的（无效值会引起运行时错误），则没有必要使用健壮形式的测试

3. 如果错误条件非常重要，则进行健壮形式的测试是合适的。

4. 如果输入数据以离散值区间和集合定义，则等价类测试是合适的。当然也适用于如果变量值越界系统就会出现故障的系统

5. 通过结合边界值测试，等价类测试可得到加强。（我们可以“重用”定义等价类的工作成果。）

6. 如果程序函数很复杂，则等价类测试是被指示的，在这种情况下，函数的复杂性可以帮助标识有用的等价类，就像NextDate函数一样。

7. 强等价类测试假设变量是独立的，相应的测试用例相乘会引起冗余问题　如果存在依赖关系，则常常会生成“错误”测试用例，就像NextDate函数一样.（第7章将介绍的决策表技术可解决这个问题　）

8. 在发现“合适”的等价关系之前，可能需要进行多次尝试，就像NextDate函数例子一样。在其他情况下，存在“明显”或“自然”等价关系。如果不能肯定，最好对任何合理的实现进行再次预测。

9. 强和弱形式的等价类测试之间的差别，有助于区分累进测试和回归测试。

6.6 参考文献

Mosley, Daniel J., *The Handbook of MIS Application Software Testing,* Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1993.

Myers, Glenford J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.

6.7 练习

1. 请以NextDate函数的36个强一般等价类测试用例为基础，按所讨论的那样修改日期类，然后找出其他9个测试用例。

2. 如果使用强类型语言编译器，请讨论怎样才能执行健壮等价类测试用例。

3. 请针对包含了直角的扩展三角形问题来修改弱一般等价类集合。

4. 请对比单/多缺陷假设与边界值测试和等价类测试。

5. 对电话账单来说，春季和秋季的标准时间与夏时制时间的转换会带来有意思的问题　春季，这种转换发生在（3月末、4月初的）星期日凌晨2:00时，这时时钟要设置为凌晨3:00时。对称的转换通常发生在10月最后一个星期日，时钟要从2:59:59调回到2:00:00。

请为采用以下费率记账的长途电话服务函数开发等价类：

通话时间 < = 20分钟时，每分钟收费0.05美元，通话时间不到1分种时按1分钟计算.

通话时间 > 20分钟时，收费1.00美元，外加每分钟0.10美元，超过20分钟的部分、不到1分钟时按1分钟计算.

假设：

- 通话计费时间从被叫方应答开始计算，到呼叫方挂机时结束。
- 通话时间的秒数四舍五入到分钟.
- 没有超过20个小时的通话。

6. 如果读者做了第2章的练习8、第5章的练习5，就会熟悉CRC出版公司网站

（http://www.crcpress.com）的下载过程 通过这个网站，可以下载叫做BlackBox.exe的可执行Visual Basic程序。（它有一个Naive.exe面向文件的版本，也包含同样的插入缺陷）文件A2.txt、B2.txt和C2.txt分别包含三角形、NextDate和佣金问题的等价类测试用例，请运行这些测试用例集合，在读者自己命名的文件中保存测试结果，并与第2章自然测试和第5章边界值测试的结果进行比较。

第7章 基于决策表的测试

在所有功能性测试方法中，基于决策表的测试方法是最严格的，因为决策表具有逻辑严格性。人们使用两种密切关联的方法：因果图法（Elmendorf，1973；Myers，1979）和决策表格法（Mosley，1993）。与决策表相比，这两种方法使用起来更麻烦，并且全冗余。这两种方法在Mosley（1993）中都做了介绍。

7.1 决策表

自从20世纪60年代初以来，决策表一直被用来表示和分析复杂逻辑关系。决策表很适合描述不同条件集合下采取行动的若干组合的情况。表7-1给出了基本决策表术语

表7-1 决策表的各个部分

桩	规则1	规则2	规则3、4	规则5	规则6	规则7、8
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	—	T	F	—
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

决策表有四个部分：粗竖线的左侧是桩部分；右侧是条目部分。横粗线的上面是条件部分，下面是行动部分。因此，我们可以引用条件桩、条件条目、行动桩和行动条目。条目部分中的一列是一条规则。规则指示在规则的条件部分中指示的条件环境下要采取什么行动。在图7-1给出的决策表中，如果c1、c2和c3都为真，则采取行动a1和a2。如果c1和c2都为真而c3为假，则采取行动a1和a3。在c1为真c2为假条件下，规则中的c3条目叫做“不关心”条目。不关心条目有两种主要解释：条件无关或条件不适用。有时人们用“不适用（n/a）”表示这后一种解释。

如果有二叉条件（真/假，是/否，0/1），则决策表的条件部分是旋转了90度的（命题逻辑）真值表。这种结构能够保证我们考虑了所有可能的条件值组合。如果使用决策表标识测试用例，那么决策表的这种完备性质能够保证一种完备的测试。所有条件都是二叉条件的决策表叫做有限条目决策表。如果条件可以有多个值，则对应的决策表叫做扩展条目决

策表。我们将给出NextDate问题的两种决策表例子。

决策表被设计为说明性的（与命令性相反），给出的条件没有特别的顺序，而且所选择的行动发生时也没有任何特定顺序。

表示方法

为了使用决策表标识测试用例，我们把条件解释为输入，把行动解释为输出。有时条件最终引用输入的等价类，行动引用被测软件的主要功能处理部分。这时规则就解释为测试用例。由于决策表可以机械地强制为完备的，因此可以有测试用例的完整集合。

有多种产生决策表的方法对测试人员更有用。一种很有用的风格是增加行动，以显示什么时候规则在逻辑上不可能满足。

在表7-2所示的决策表中，给出了不关心条目和不可能规则使用的例子。正如第一条规则所指示，如果整数a、b和c不构成三角形，则我们根本不关心可能的相等关系。在规则2、3和4中，如果两对整数相等，则根据传递性，第三对整数也一定相等，因此第一条规则的否定条目使这些规则不可能满足。

表7-2 三角形问题决策表

c1：a、b、c构成三角形？	N	Y	Y	Y	Y	Y	Y	Y	Y
c2：a = b？	—	Y	Y	Y	Y	N	N	N	N
c3：a = c？	—	Y	Y	N	N	Y	Y	N	N
c4：b = c？	—	Y	N	Y	N	Y	N	Y	N
a1：非三角形	X								
a2：不等边三角形									X
a3：等腰三角形					X		X	X	
a4：等边三角形		X							
a5：不可能			X	X		X			

表7-3所示的决策表给出了有关表示方法的另一种考虑：条件的选择可以大大地扩展决策表的规模。这里将老条件（c1：a、b、c构成三角形？）扩展为三角形特性的三个不等式的详细表示。如果有一个不等式不成立，则三个整数就不能构成三角形。我们还可以进一步扩展，因为不等式不成立有两种方式：一条边等于另外两条边的和，或严格大于另外两条边的和。

表7-3 经过修改的三角形问题决策表

c1：a < b + c？	F	T	T	T	T	T	T	T	T	T	T
c2：b < a + c？	—	F	T	T	T	T	T	T	T	T	T
c3：c < a + b？	—	F	T	T	T	T	T	T	T	T	
c4：a = b？	—	—	—	T	T	T	T	F	F	F	F
c5：a = c？		—		T	T	F	F	T	T	F	F
c6：b = c？	—		—	T	F	T	F	T	F	T	F

（续）

a1：非三角形	X	X	X								
a2：不等边三角形											X
a3：等腰三角形							X		X	X	
a4：等边三角形				X							
a5：不可能					X	X		X			

如果条件引用了等价类，则决策表会有一种典型的外观。表7-4所示的决策表来自NextDate问题，引用了可能的月份变量相互排斥的可能性。由于一个月份就是一个等价类，因此不能有两个条目同时为真的规则。不关心条目（－）的实际含义是“必须失败”。有些决策表使用者用F!表示这一点。

表7-4　带有相互排斥条件的决策表

条件	规则1	规则2	规则3
c1：月份在M1中?	T	—	—
c2：月份在M2中?	—	T	—
c3：月份在M3中?	—	—	T
a1			
a2			
a3			

不关心条目的使用，对完整决策树的识别方式有微妙的影响。对于有限条目决策表，如果有n个条件，则必须有 2^n 条规则。如果不关心条目实际地表明条件是不相关的，则可以按以下方法统计规则数。没有不关心条目的规则统计为1条规则；规则中每出现一个不关心条目，该规则数乘一次2。表7-3所示决策表的规则条数统计如表7-5所示。请注意，规则总数是64（正好是应该得到的规则条数）。

表7-5　表7-3所示规则条数统计的决策表

c1：a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2：b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c3：c < a + b?	—	—	F	T	T	T	T	T	T	T	T
c4：a = b?	—	—	—	T	T	T	T	F	F	F	F
c5：a = c?	—	—	—	T	T	F	F	T	T	F	F
c6：b = c?	—	—	—	T	F	T	F	T	F	T	F
规则条数统计	32	16	8	1	1	1	1	1	1	1	1
a1：非三角形	X	X	X								
a2：不等边三角形											X
a3：等腰三角形							X		X	X	
a4：等边三角形				X							
a5：不可能					X	X		X			

如果将这种简化算法应用于表7-4所示的决策表，会得到如表7-6所示的规则条数统计。

表7-6　带有相互排斥条件的决策表规则条数统计

条件	规则1	规则2	规则3
c1：月份在M1中？	T	—	—
c2：月份在M2中？	—	T	—
c3：月份在M3中？	—	—	T
规则条数统计	4	4	4
a1			

应该只有八条规则，所以显然有问题。为了找出问题所在，我们扩展所有三条规则，用可能的T或F替代“－”，如图7-7所示。

表7-7　表7-6的扩展版本

条件	1.1	1.2	1.3	1.4	2.1	2.2	2.3	2.4	3.1	3.2	3.3	3.4
c1：月份在M1中？	T	T	T	T	T	T	F	F	T	T	F	F
c2：月份在M2中？	T	T	F	F	T	T	T	T	T	F	T	F
c3：月份在M3中？	T	F	T	F	T	F	T	F	T	T	T	T
规则条数统计	1	1	1	1	1	1	1	1	1	1	1	1
a1												

请注意，所有条目都是T的规则有三条：规则1.1、2.1和3.1；条目是T、T、F的规则有两条：规则1.2和2.2。类似地，规则1.3和3.2、2.3和3.3也是一样的。如果去掉这种重复，最后可得到七条规则，缺少的规则是所有条件都是假的规则。这种处理的结果如表7-8所示，表中还给出了不可能出现的规则。

表7-8　包含不可能出现的规则的相互排斥条件

条件	1.1	1.2	1.3	1.4	2.3	2.4	3.4	
c1：月份在M1中？	T	T	T	T	F	F	F	F
c2：月份在M2中？	T	T	F	F	T	T	F	F
c3：月份在M3中？	T	F	T	F	T	F	T	F
规则条数统计	1	1	1	1	1	1	1	1
a1：不可能	X	X	X		X			X

这种识别（和开发）完备决策表的能力，使我们在解决冗余性和不一致性方面处于很有利的地位。表7-9给出的决策表是冗余的，因为有三个条件和九条规则（规则9与规则4相同）。

表7-9　一个冗余决策表

条件	1~4	5	6	7	8	9
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	X
a2	—	X	X	X	—	
a3	X	—	X	X	X	X

请注意，规则9的行为条目与规则1~4的条目相同。只要冗余规则中的行为与决策表相应的部分相同，就不会有什么大问题。如果行为条目不同，例如表7-10所示的情况，则会遇到比较大的问题。

表7-10　一个不一致的决策表

条件	1~4	5	6	7	8	9
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	
a2	—	X	X	X	—	X
a3	X	—	X	X	X	

如果表7-10所示的决策表被用来处理事务，其中c1是真，c2和c3都是假，则规则4和规则9都适用。我们可以观察到两点：

1. 规则4和规则9是不一致的。
2. 决策表是非确定的。

规则4和规则9是不一致的，因为行为集合是不同的。整个决策表是不确定的，因为不能确定是应用规则4还是应用规则9。测试人员的基本原则是在决策表中小心使用不关心条目

7.2　三角形问题的测试用例

使用表7-3给出的决策表，可得到11个功能性测试用例：3个不可能测试用例，3个测试用例违反三角形性质，1个测试用例可得到等边三角形，1个测试用例可得到不等边三角形，3个测试用例可得到等腰三角形（如表7-11所示）。如果扩展决策表以显示两种违反三角形性质的方式，可以再选三个测试用例（一条边正好等于另外两条边的和）。做到这一点需要做一定的判断，因为否则规则会呈指数级增长。在这种情况下，最终会再得到很多不关心条目和不可能的规则。

表7-11　根据表7-3得到的测试用例

用例ID	a	b	c	预期输出
DT1	4	1	2	非三角形
DT2	1	4	2	非三角形
DT3	1	2	4	非三角形
DT4	5	5	5	等边三角形
DT5	?	?	?	不可能
DT6	?	?	?	不可能
DT7	2	2	3	等腰三角形
DT8	?	?	?	不可能
DT9	2	3	2	等腰三角形
DT10	3	2	2	等腰三角形
DT11	3	4	5	不等边三角形

7.3　NextDate函数测试用例

选择NextDate函数，是因为它可以说明输入定义域中的依赖性问题，这使得这个例子成为基于决策表测试的一个完美例子，因为决策表可以突出这种依赖关系。第6章曾经标识出NextDate函数输入定义域中的等价类。第6章发现的限制之一，是从等价类中随意地选取输入值，会产生“很奇怪”的测试用例，例如找出1812年6月31日的下一天。问题的根源是假设变量都是独立的。如果变量确实是独立的，则使用类的笛卡儿积是有意义的。如果变量之间在输入定义域中存在逻辑依赖关系，则这些依赖关系在笛卡儿积中就会丢失（说抑制可能更确切）。决策表格式通过使用“不可能行动”概念表示条件的不可能组合，使我们能够强调这种依赖关系。本节将对NextDate函数的决策表描述做三遍尝试。

7.3.1　第一次尝试

标识合适的条件和行动，代表测试工艺化的一个机会。假设首先从接近第6章中使用过的等价类集合开始。

M1 = {月份：每月有30天}
M2 = {月份：每月有31天}
M3 = {月份：此月是2月}
D1 = {日期：1≤日期≤28}
D2 = {日期：日期 = 29}
D3 = {日期：日期 = 30}
D4 = {日期：日期 = 31}
Y1 = {年：年是闰年}
Y2 = {年：年不是闰年}

如果我们希望突出不可能的组合，则可以建立具有以下条件和行动的有限条目决策表（请注意，年变量对应的等价类收缩为表7-12中的一个条件。）

表7-12 有256条规则的第一次尝试

条件			
c1：月份在M1中？	T		
c2：月份在M2中？		T	
c3：月份在M3中？			T
c4：日期在D1中？			
c5：日期在D2中？			
c6：日期在D3中？			
c7：日期在D4中？			
c8：年在Y1中？			
a1：不可能			
a2：NextDate			

这个决策表会有256条规则，其中很多是不可能的。如果要显示为什么这些规则是不可能的，可以将行动修改为：

a1：月份中的天太多。

a2：不能出现在非闰年中。

a3：计算NextDate。

7.3.2 第二次尝试

如果我们将注意力集中到NextDate函数的闰年问题上，则可以使用第6章介绍过的等价类集合。这些类有一个包含36个三元组的笛卡儿积，其中有不可能的等价类。

为了说明另一种决策表表示方法，这一次采用扩展条目决策表开发，并更仔细地研究行动桩。在构建扩展条目决策表时，必须保证等价类构成输入定义域的真划分。（第3章曾经介绍过，划分是一组不相交的子集，子集的并是全集。）如果规则条目之间存在"重叠"，则会存在冗余情况，使得多个规则都能够满足。这里，Y2是一组1812~2012之间的年份，并除以4，2000年除外。

M1 = {月份：每月有30天}

M2 = {月份：每月有31天}

M3 = {月份：此月是2月}

D1 = {日期：1 ≤ 日期 ≤ 28}

D2 = {日期：日期 = 29}

D3 = {日期：日期 = 30}

D4 = {日期：日期 = 31}

Y1 = {年：年 = 2000}

Y2 = {年：年是闰年}

Y3 = {年：年是平年}

从某种意义上说，我们采用的是“灰盒”技术，因为更仔细地研究了NextDate函数。为了产生给定日期的NextDate，能够使用的操作只有五种：日期和月份的增1和复位、年的增1（我们不允许通过复位年来回退时间）。

这些条件可以产生有对应等价类笛卡儿积的36个规则的决策表。结合不关心条目，可得到表7-13所示的有17条规则的决策表。仍然存在逻辑不可能的规则，但是这个表有助于我们标识测试用例的扩展输出。如果填满这个决策表的行动条目，就会发现12月有一些麻烦的问题（规则8）。我们下面解决这些问题。

表7-13 有36条规则的第二次尝试

	1	2	3	4	5	6	7	8
c1：月份在	M1	M1	M1	M1	M2	M2	M2	M2
c2：日期在	D1	D2	D3	D4	D1	D2	D3	D4
c3：年在	—	—	—	—	—	—	—	
规则条数统计	3	3	3	3	3	3	3	3
行为								
a1：不可能				X				
a2：日期增1	X	X			X	X	X	
a3：日期复位			X					X
a4：月份增1			X					[illegible]
a5：月份复位								[illegible]
a6：年增1								[illegible]

	9	10	11	12	13	14	15	16	17
c1：月份在	M3	M3	M3	M3	M3	M3	M3	M3	M3
c2：日期在	D1	D1	D1	D2	D2	D2	D3	D3	D4
c3：年在	Y1	Y2	Y3	Y1	Y2	Y3			
规则条数统计	1	1	1	1	1	1	3	3	3
行为									
a1：不可能						X	X	X	X
a2：日期增1		X							
a3：日期复位	X		X	X	X				
a4：月份增1	X		X	X	X				
a5：月份复位									
a6：年增1									

7.3.3 第三次尝试

通过引入等价类的第三个集合，可以澄清年末问题。这一次可以特别关注日和月，并重新使用第一次尝试的较简单的闰年或非闰年条件，因此2000年没有特别处理。（本来还可以作第四次尝试，采用第二次尝试的年等价类，但是现在读者已经掌握要点了。）

M1 = {月份：每月有30天}

M2 = {月份：每月有31天，12月除外}

M3 = {月份：此月是2月}

M4 = {月份：此月是12月}

D1 = {日期：1≤日期≤27}

D2 = {日期：日期 = 28}

D3 = {日期：日期 = 29}

D4 = {日期：日期 = 30}

D5 = {日期：日期 = 31}

Y1 = {年：年是闰年}

Y2 = {年：年不是闰年}

这个等价类的笛卡儿积包含40个元素。所产生的组合规则包含不关心条目，如表7-14所示，可与第二次的36条规则比较。第1章曾经提到过，大的测试用例集合是否一定比小的测试用例集合好。这里我们有一个22条规则的决策表，得到的NextDate函数的描述比包含36条规则的决策表更清晰。头5条规则处理有30天的月份，请注意，这里不考虑闰年。接下来的两组规则（规则6~10、规则11~15）处理有31天的月份，头5条规则处理12月之外的月份，后5条规则处理12月。不可能规则也在决策表中列出，尽管存在一些高效测试人员可能会有疑问的冗余10条规则中的8条只是对日期增1。针对这个子功能是否真的需要8条单独的测试用例吗？可能不需要，但是请注意我们可以通过决策表得到的启发。最后的7条规则关注的是2月和闰年。

表7-14 NextDate函数的决策表

	1	2	3	4	5	6	7	8	9	10
c1：月份在	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2
c2：日期在	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5
c3：年在	—	—	—	—	—	—	—	—	—	—
行为										
a1：不可能					X					
a2：日期增1	X	X	X			X	X	X	X	
a3：日期复位				X						X
a4：月份增1				X						X
a5：月份复位										
a6：年增1										

（续）

	11	12	13	14	15	16	17	18	19	20	21	22
c1：月份在	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
c2：日期在	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
c3：年在	—				-	—	Y1	Y2	Y1	Y2	-	
行为												
a1：不可能										X	X	X
a2：日期增1	X	X	X	X		X	X					
a3：日期复位					X			X	X			
a4：月份增1								X	X			
a5：月份复位					X							
a6：年增1					X							

表7-14所示的决策表是第2章NextDate函数源代码的基础。这个例子从另一个方面说明测试如何能够很好地改进程序设计。所有决策表分析都应该在NextDate函数的详细设计期间完成。

我们可以使用决策表代数进一步简化这22个测试用例。如果决策表中两个规则的行动集合相同，则一定至少有一个条件能够把两条规则用不关心条目合并。这正体现出决策表等价于用于标识等价类的“相同处理”方针。在某种意义上，我们就是在标识规则的等价类。例如，规则1、2和3涉及有30天的月份的日期类D1、D2和D3。类似地，有31天的月份的日期类D1、D2、D3和D4也可以合并，2月的D4和D5也可以合并。所得到的结果如表7-15所示。

表7-15 NextDate函数的精简决策表

	1~3	4	5	6~9	10
c1：月份在	M1	M1	M1	M2	M2
c2：日期在	D1、D2、D3	D4	D5	D1、D2、D3、D4	D5
c3：年在		—	--	--	--
行为					
a1：不可能			X		
a2：日期增1	X			X	
a3：日期复位		X			X
a4：月份增1		X			X
a5：月份复位					
a6：年增1					

	11~14	15	16	17	18	19	20	21、22
c1：月份在	M3	M3	M4	M4	M4	M4	M4	M4
c2：日期在	D1、D2、D3、D4	D5	D1	D2	D2	D3	D3	D4、D5
c3：年在	-	--		Y1	Y2	Y1	Y2	
行为								
a1：不可能							X	X
a2：日期增1	X		X	X				
a3：日期复位		X			X	X		
a4：月份增1					X	X		
a5：月份复位		X						
a6：年增1		X						

相应的测试用例如表7-16所示。

表7-16 NextDate函数的决策表测试用例

用例ID	月份	日期	年	预期输出
1~3	4	15	2001	2001年4月16日
4	4	30	2001	2001年5月1日
5	4	31	2001	不可能
6~9	1	15	2001	2001年1月16日
10	1	31	2001	2001年2月1日
11~14	12	15	2001	2001年12月16日
15	12	31	2001	2002年1月1日
16	2	15	2001	2001年2月16日
17	2	28	2004	2004年2月29日
18	2	28	2001	2001年3月1日
19	2	29	2004	2004年3月1日
20	2	29	2001	不可能
21，22	2	30	2001	不可能

7.4 佣金问题的测试用例

决策表分析不太适合佣金问题，这并不奇怪，因为在佣金问题中只有很少的决策逻辑。由于等价类中的变量是真正独立的，在条件对应等价类的决策表中没有不可能规则。因此，我们得到的测试用例与等价类测试用例一样。

7.5 指导方针与观察

与其他测试技术一样，基于决策表的测试对于某些应用程序（例如NextDate函数）很有效，但是对另一些应用程序（例如佣金问题）就不值得费这么大的事。毫不奇怪，基于决策表所适用的情况都是要发生大量决策（例如三角形问题），以及在输入变量之间存在重要的逻辑关系的情况（例如NextDate函数）。

1. 决策表技术适用于具有以下特征的应用程序：

   if-then-else逻辑很突出。

   输入变量之间存在逻辑关系。

   涉及输入变量子集的计算。

   输入与输出之间存在因果关系。

   很高的圈（McCabe）复杂度（请参阅第9章）。

2. 决策表不能很好地伸缩（有n个条件的有限条目决策表有 2^n 个规则）。有多种方法可

以解决这个问题——使用扩展条目决策表、代数简化表，将大表"分解"为小表，查找条件条目的重复模式。有关更多内容，请参阅Topper（1993）。

3. 与其他技术一样，迭代会有所帮助。第一次标识的条件和行动可能不那么令人满意。把第一次得到的结果作为铺路石，逐渐改进，直到得到满意的决策表。

7.6 参考文献

Elmendorf, William R., *Cause–Effect Graphs in Functional Testing*, IBM System Development Division, Poughkeepsie, NY. TR-00.2487. 1973.

Mosley, Daniel J., *The Handbook of MIS Application Software Testing*, Yourdon Press Prentice Hall, Englewood Cliffs, NJ, 1993.

Myers, Glenford J., *The Art of Software Testing*, Wiley Interscience. New York, 1979

Topper, Andrew et al., *Structured Methods. Merging Models, Techniques, and CASE*, McGraw-Hill, New York, 1993.

7.7 练习

1. 为三角形问题中的直角三角形开发一个决策表和额外测试用例（请参阅第2章中的练习）。请注意，会有等腰直角三角形，但是边不会是整数。

2. 请为NextDate函数的"第二次尝试"开发一个决策表。在有31天的月份的月末，日总是要复位到1。对于12月之外的其他所有月份，月份都要增1，对于12月，要复位到1月，年要增1。

3. 请为前一日函数开发一个决策表（请参阅第2章中的练习）。

4. 请扩展佣金问题，考虑销售额限制的"违反"问题。请为"公司友好"版本和"销售商友好"版本开发对应的决策表和测试用例。

5. 请讨论决策表测试能够在多大程度上处理多缺陷假设问题。

6. 请为时变问题开发决策表测试用例（请参阅第6章练习5）。

第 8 章

功能性测试回顾

在前面的三章中，我们研究了很多类型的功能性测试。联系这些测试的共同线索是都把程序看做是将输入映射到输出的数学函数。采用基于边界的方法，测试用例通过输入变量的边界值域标识，并演变为四种技术——边界值分析、健壮性测试、最坏情况测试和健壮最坏情况测试。然后我们进一步研究了输入变量；通过应该从被测程序接受“相似处理”的取值，定义了等价类。我们使用了四种等价类测试——弱一般、强一般、弱健壮和强健壮。研究相似处理的目标，就是减少通过基于定义域技术生成的测试用例的绝对数量。我们在使用决策表分析程序函数的逻辑依赖关系时，又把这个问题推进一步。任何时候如果我们有多种选择，很自然地就会想知道哪种选择最好，或至少知道如何更有见地地做出选择。本章将讨论有关测试工作量、测试效率问题，然后努力掌握有效测试的关键。

8.1 测试工作量

先回到我们的工艺师比喻上。我们通常认为工艺师谙熟自己的技巧，能够非常有效地利用自己的时间。即使占用时间稍微长一点，我们也会认为时间被很好地利用。下面讨论这些对测试技术的提示。我们讨论过的功能性测试方法，在所生成的测试用例数量上和开发这些测试用例所需的工作量上都各不相同。图8-1和图8-2给出了一般趋势，不过坐标轴需要进行解释。

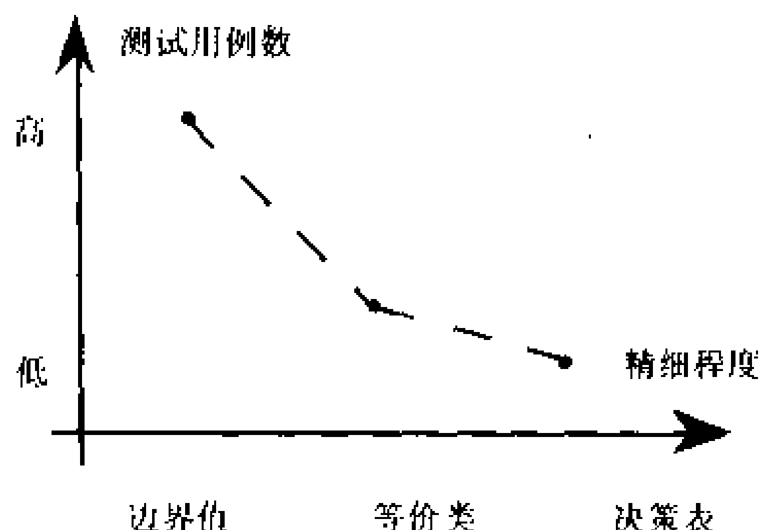


图8-1 每种测试方法的测试用例趋势线

基于定义域的技术不识别数据或逻辑依赖关系，采用非常机械的方式生成测试用例。因此，基于定义域的测试很容易被自动化。等价类技术注意到数据依赖关系和函数本身。使用这些技术需要更多的考虑，还需要更多的判断和技巧。首先要考虑如何标识等价类，之后的

处理也是机械的。决策表技术最精细，因为它要求测试人员既要考虑数据，又要考虑逻辑依赖关系。通过我们的例子可以看出，只通过一次尝试可能不能得到决策表的条件，但是如果有了一个良好的条件集合，所得到的测试用例就是完备的，在一定意义上讲还是最少的。

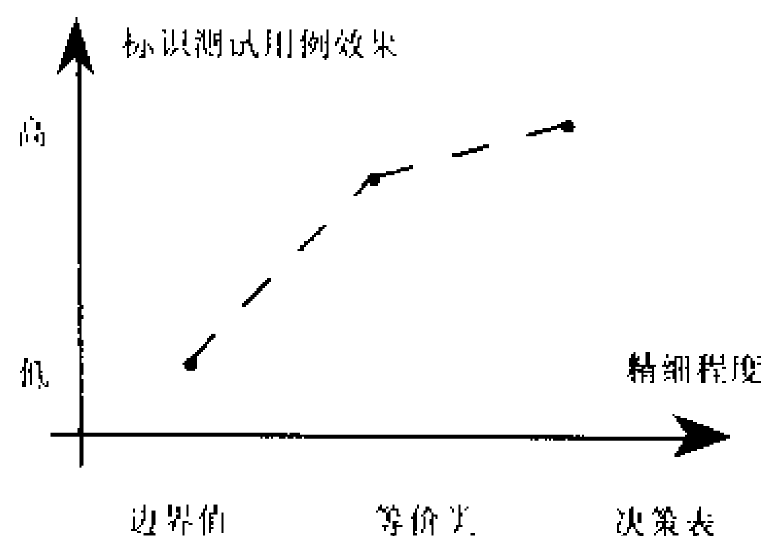


图8-2　每种测试方法的测试用例标识工作量趋势线

最终结果是在测试标识工作量和测试执行工作量做令人满意的折衷：容易使用的方法会生成大量测试用例，因此执行时间很长。如果将工作量投入到更精细的测试方法，则执行时间就会缩短。这一点非常重要，因为测试一般要执行多次。大家也许已经注意到，通过测试用例的绝对数量判断测试质量，具有与通过代码行数判断程序设计生产率类似的缺点。

我们的例子具有图8-1和图8-2所示的趋势。以下三张图（图8-3~图8-5）来自归纳以各种方法针对我们的例子所生成的测试用例数量的电子表格。对于基于定义域的测试方法，所生成的测试用例个数相同，这既反映了这些技术具有相同的机械性质，又反映了描述每种方法所生成的测试用例个数的公式近似。在强等价类测试和决策表测试之间存在主要差别，这反映出问题的逻辑复杂性，因此应该看到它们的差别。当我们研究结构性测试（第9章和第10章）时，会看到这种差别对于测试具有重要意义。这三张图通过图8-6叠加在一起。

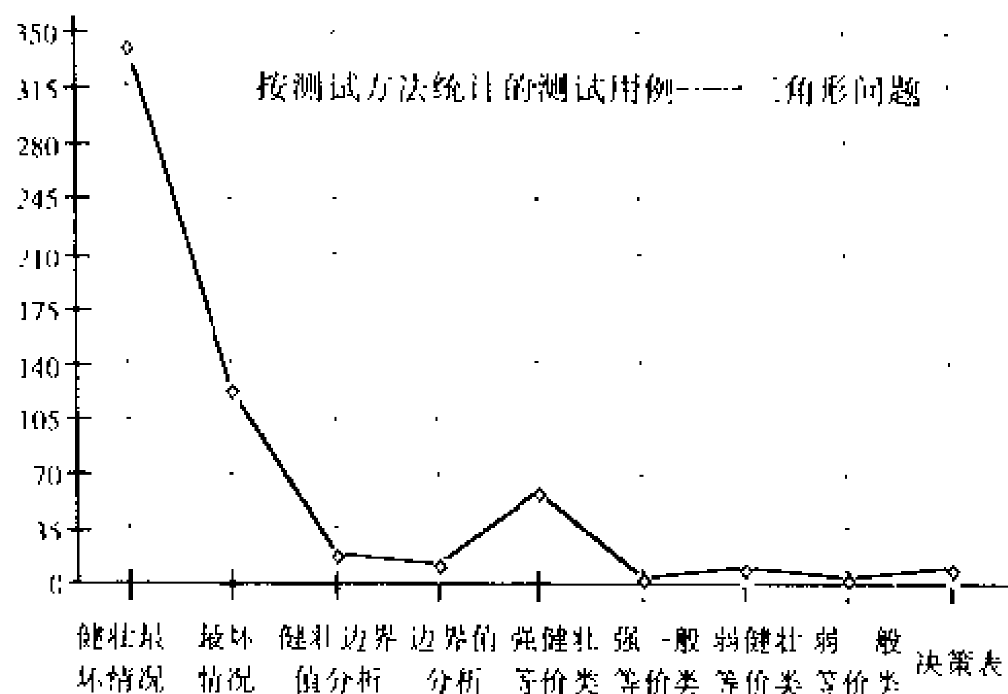


图8-3　三角形问题的测试用例趋势线

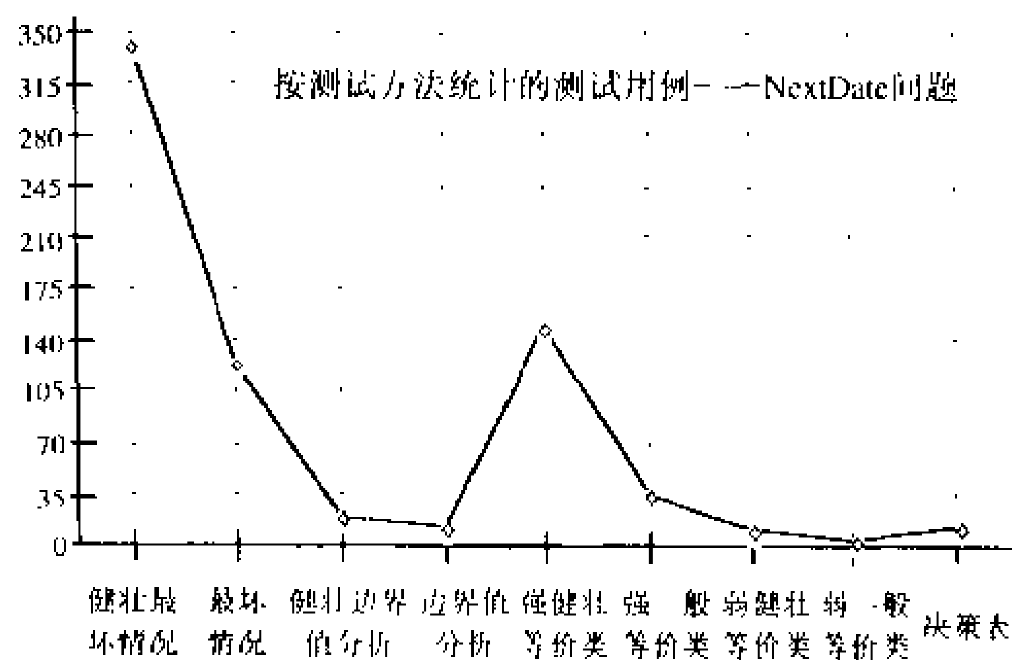


图8-4　NextDate问题的测试用例趋势线

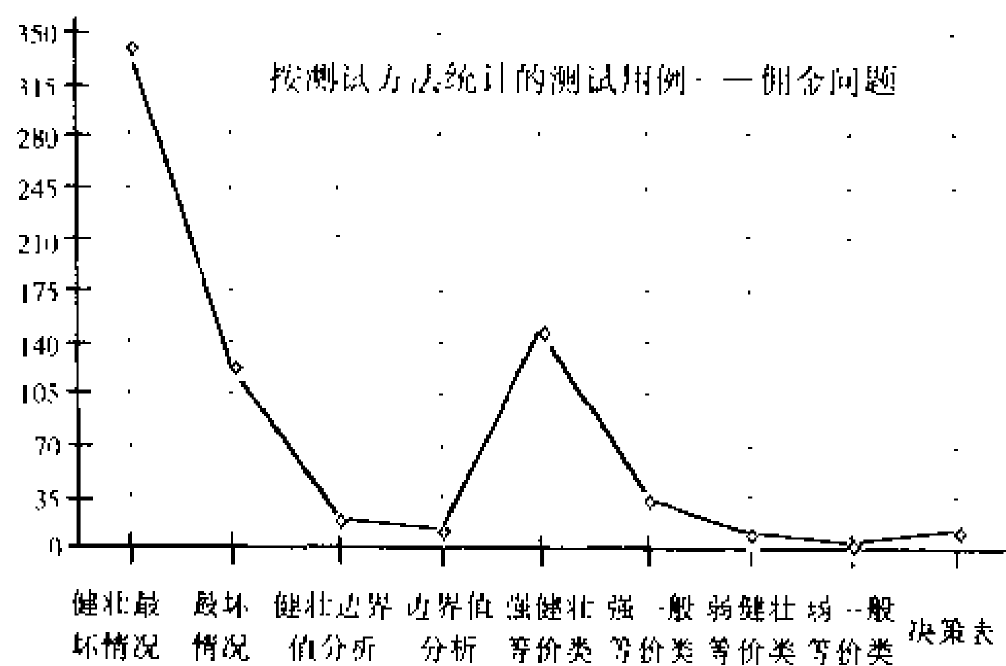


图8-5　佣金问题的测试用例趋势线

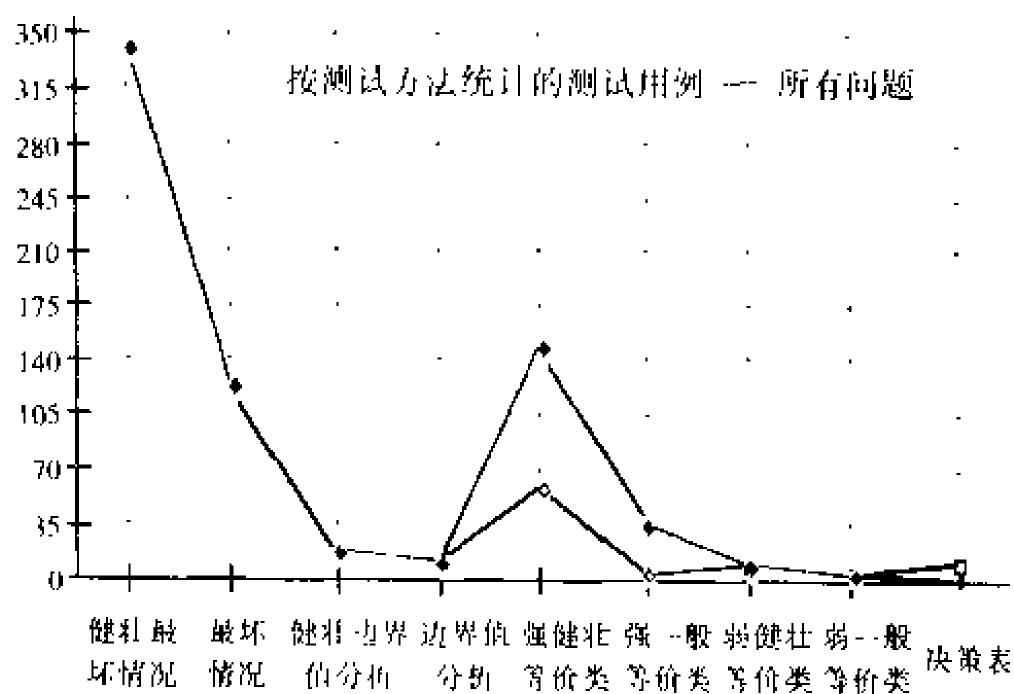


图8-6　三个问题的测试用例趋势线

8.2 测试效率

如果仔细研究这些测试用例集合，就会体会到功能性测试的基本局限：未测试的功能漏洞和冗余测试。例如NextDate问题，决策表（通过三次尝试使其完善）产生了13个测试用例。我们可以确信这些测试用例（在某种意义上）是完备的，因为决策表是完备的。另一方面，最坏情况边界值分析产生125个测试用例。仔细研究这些测试用例，就会发现结果并不令人满意：通过测试用例1~5预期能够得到什么结论？这些测试用例通过五个不同年份中的1月1日检查NextDate问题。年对日历的这个部分没有关系，因此我们预期这种测试用例有一个就足够的。如果大致估计有"10倍"的冗余，那么可以预期压缩到25个测试用例，非常接近决策表方法的22。再进一步研究，发现有多个2月测试，但是都没有涉及28和29日，并且与闰年也没有联系。我们不仅有10倍的冗余，而且围绕2月底和闰年，测试还有严重漏洞。

强等价类测试用例的前进方向正确：有36个测试用例，其中有11个是不可能的。同样，这些不可能测试用例的根源，也是等价类之间的独立性假设。除了6个之外（表7-16中的测试用例2、7、12、15、17和18），决策表中的所有测试用例都可映射到相应的强等价类测试用例（请参阅第6.3.1节中的表）上。这些测试用例中的一半用于处理非2月的28日问题，因此没有多大意思。其余三个测试用例很有用，因为它们测试了强等价类测试用例遗漏的可能性。所有这些分析都支持两点结论：功能性测试有漏洞，使用更精细的手段能够缩小这些漏洞。

可以通过量化测试效率把这个问题推进一步吗？在直觉上，一组没有漏洞没有冗余的测试用例可以用于量化测试效率。可以计算A方法所生成的测试用例总数与B方法生成的测试用例总数的比值，或与某个测试用例基础的比值。这样做通常工作量很大而效果很小，但是有时管理层要求提供数字，即使这些数字没有多少实际意义。在讨论完结构性测试之后，第11章还要讨论这个问题。结构性测试方法支持有意义（并且有用）的指标，通过这些指标能够量化测试效率要好得多。同时，通过对测试用例增加简短用途注释，有助于识别冗余。如果发现多个测试用例具有同样的用途，则可能（正确地）发现冗余。发现漏洞要更困难一些：如果只能使用功能性测试，则我们所能够做的就是比较通过两种方法得到的测试用例。一般来说，更精细的方法有助于识别漏洞，但是却不能保证什么。我们可以为某个程序开发出优秀的强等价类，然后又构造出很差的决策表。

8.3 测试的有效性

关于测试用例集合，我们真正想知道的是它们的测试效果如何，但是需要澄清"有效性"的含义。最简单的方法很教条：指定一种方法，用这种方法生成测试用例，然后运行测试用例。这种方法是绝对的，一致性是可度量的，因此可以用做构造性兼容的基础。通过放松教条式的指定，要求测试人员利用本书每章末尾的指导方针自选"合适的方法"，可以改

进这种判别方法。通过合适的混合方法，就像在第5章中处理佣金问题一样，还可以得到另外一种渐进式的改进。

结构性测试技术产生判断测试有效性的第二种选择。第9章将讨论程序执行路径概念，这类概念可以提供评价测试有效性的很好手段。我们将能够通过所经过的执行路径检查测试用例集合。如果特定的路径被经过多次，则冗余性就可能存在疑问。有时这种冗余可能是故意设计的，第10章还将讨论这个问题。

测试有效性的最佳解释是最困难的（这一点不那么令人奇怪）。我们真正想知道的是一组测试用例能够怎样有效地找出程序中的缺陷。出现这种困难有两方面的原因：首先，这要假定我们知道程序中的所有缺陷。形成死循环的是，如果我们知道程序中的所有缺陷，就会采取有针对性的措施。由于我们不知道程序中的所有缺陷，因此永远也不会知道给定方法所产生的测试用例是否能够发现这些缺陷。第二个原因更有理论性：假设无缺陷的程序与计算机科学中著名的死机问题等效，这被认为是不可能的。我们所能够做的，就是根据不同类型的缺陷进行反向研究。给出特定的一种缺陷，我们可以选择最有可能发现这种缺陷的测试方法（功能性测试和结构性测试）。如果结合最可能出现的缺陷种类的知识，最终会得到可提高措施有效性的实用方法。通过跟踪所开发软件中的缺陷的种类（和密度），还可以改进这种方法

8.4　指导方针

以下是一段我很喜欢的测试故事。有一个醉汉在路灯下的人行道上爬行，当警察问他在干什么时，他说他在找汽车钥匙。“你是在这里丢的吗？”警察问。“不，我在停车场丢的，但是这里的光线更亮。”

这个小故事对测试人员具有重要启示：测试不大可能存在的缺陷是没有意义的。很好地了解最有可能发生的缺陷（或损害）种类，然后选择最有可能发现这类缺陷的测试方法，这样是更为有效的。

很多时候，我们甚至对可能普遍存在的缺陷没有一点感觉。怎么办呢？我们能够采用的最好办法是利用程序的已知属性，选择处理这种属性的方法，有点像“依罪量刑”的味道。在选择功能性测试方法时很有用的属性包括：

变量是否表示物理量或逻辑量？
在变量之间是否存在依赖关系？
是假设单缺陷，还是假设多缺陷？
是否有大量例外处理？

以下是功能性测试技术选择的初步的“专家系统”：

1. 如果变量引用的是物理量，可采用定义域测试和等价类测试
2. 如果变量是独立的，可采用定义域测试和等价类测试
3. 如果变量不是独立的，可采用决策表测试
4. 如果可保证是单缺陷假设，可采用边界值分析和健壮性测试
5. 如果可保证是多缺陷假设，可采用最坏情况测试、健壮最坏情况测试和决策表测试
6. 如果程序包含大量例外处理，可采用健壮性测试和决策表测试
7. 如果变量引用的是逻辑量，可采用等价类测试用例和决策表测试

还会出现这些条件的组合情况，因此，我们将上述指导方针作为决策表在表8-1中进行归纳

表8-1　功能性测试的合适选择

c1	变量（P，物理；L，逻辑）	P	P	P	P	P	L	L	L	L	L
c2	是独立变量吗？	Y	Y	Y	Y	N	Y	Y	Y	Y	N
c3	是单缺陷假设吗？	Y	Y	N	N	--	Y	Y	N	N	
c4	有大量例外处理吗？	Y	N	Y	N		Y	N	Y	N	
a1	边界值分析		x								
a2	健壮性测试	x									
a3	最坏情况测试				x						
a4	健壮最坏情况测试			x							
a5	传统等价类测试	x		x			x		x		
a6	弱等价类测试	x	x				x	x			
a7	强等价类测试			x	x	x			x	x	x
a8	决策表测试					x					x

8.5 案例研究

通过以下这个例子，我们可以比较功能性测试方法，并运用前面给出的指导方针。一个假想的保险金计算程序，根据两个因素计算半年保险金：投保人的年龄和驾驶历史记录：

$$保险金 = 基本保险费率 \times 年龄系数 - 安全驾驶折扣$$

年龄系数是投保人年龄的函数，如果投保人驾驶执照上的当前点数（根据交通违规次数确定）低于与年龄有关的门限，则给予安全驾驶折扣。书面保险政策的驾驶人年龄范围为从16~100岁，如果投保人有12点，则驾驶人的执照就会被吊销（因此不需要保险）。基本保险费率随时间变化，对于这个例子，是每半年500美元。

年龄范围	年龄系数	门限点数	安全驾驶折扣
16≤年龄<25	2.8	1	50
25≤年龄<35	1.8	3	50
35≤年龄<45	1.0	5	100
45≤年龄<60	0.8	7	150
60≤年龄<100	1.5	5	200

根据输入变量年龄和点数的最坏情况边界值测试产生以下极端值，所对应的25个测试用例如图8-7所示。

变量	最小值	略大于最小值	正常值	略小于最大值	最大值
年龄	16	17	54	99	100
点数	0	1	6	11	12

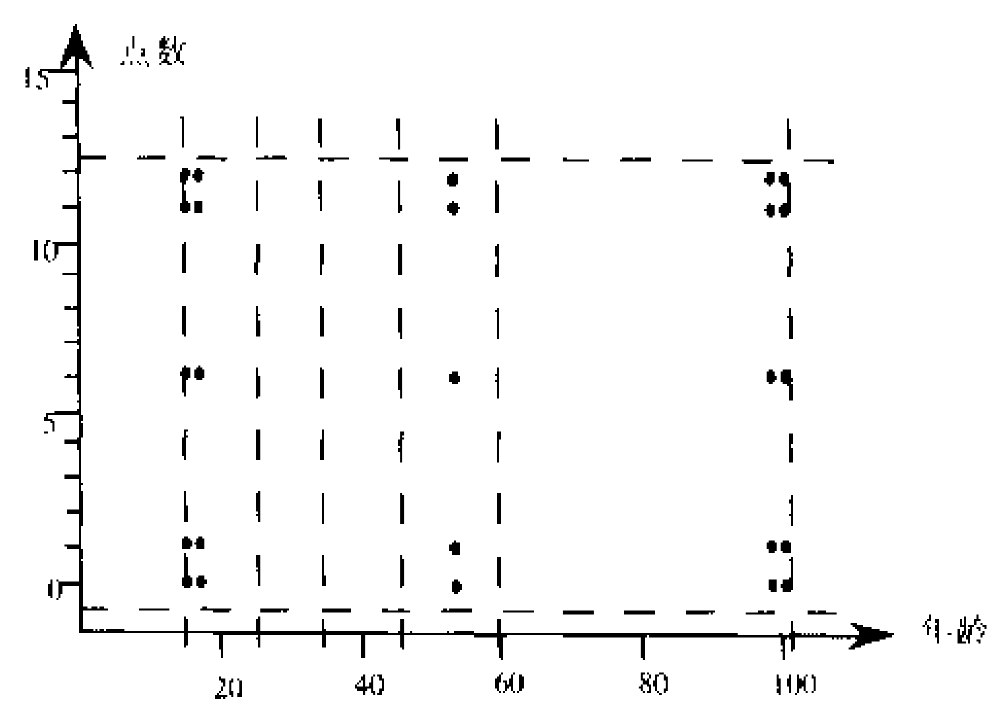


图8-7 保险金计算程序的最坏情况边界值测试

我认为不是所有人都会对这些测试用例满意，因为遗漏了太多的问题陈述。没有测试各种年龄门限，也没有测试点数门限。通过更仔细地考虑年龄范围和点数范围，可以改进测试用例集合。

A1 = {年龄：16≤年龄<25}

A2 = {年龄：25≤年龄<35}

A3 = {年龄：35≤年龄<45}

A4 = {年龄：45≤年龄<60}

A5 = {年龄：60≤年龄<100}

P1 = {点数 = 0，1}

P2 = {点数 = 2，3}

P3 = {点数 = 4，5}

P4 = {点数 = 6，7}

P5 = {点数 = 8，9，10，11，12}

由于这些范围在“端点”汇合，因此可以得到如表8-2所示的最坏情况测试值。请注意，点数变量的离散值本身不能产生某些案例的略大于最小值、略小于最大值约定。

表8-2　详细最坏情况值

变量	最小值	略大于最小值	正常值	略小于最大值	最大值
年龄	16	17	20	24	
年龄	25	26	30	34	
年龄	35	36	40	44	
年龄	45	46	53	59	
年龄	60	61	75	99	100
点数	0	—	—		1
点数	2	—	—	—	3
点数	4	—	—	—	5
点数	6	—		-	7
点数	8	9	10	11	12

如果画出网格，可以得到类似图8-8给出的内容。每个竖向集合（其中，年龄变量保持不变）有13个点，年龄变量的每个值（共有21个值）都有这样的一列，因此共有273个最坏情况边界值测试用例。这显然会存在严重冗余，因此需要引入等价类测试。

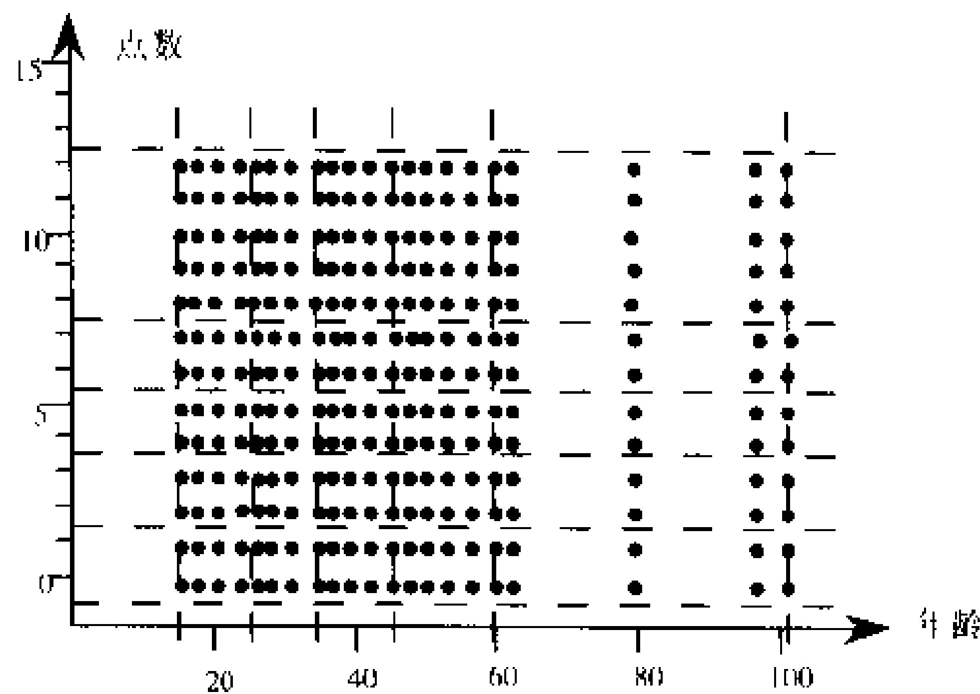


图8-8　保险金计算程序的详细最坏情况边界值测试用例

年龄集合A1~A5和点数集合P1~P5，是等价类的自然选择。相应的等价类测试用例如图8-9所示，空心圆点对应强一般测试用例，实心圆点对应弱一般测试用例。

等价类测试明显可以缓解冗余问题，但是看起来仍然还有改进余地。为什么要针对A1

测试所有点数类P2~P5？一旦超过点数门限，安全驾驶折扣就没有了。我们可以通过表8-3所示的扩展条目决策表解决这类依赖关系。

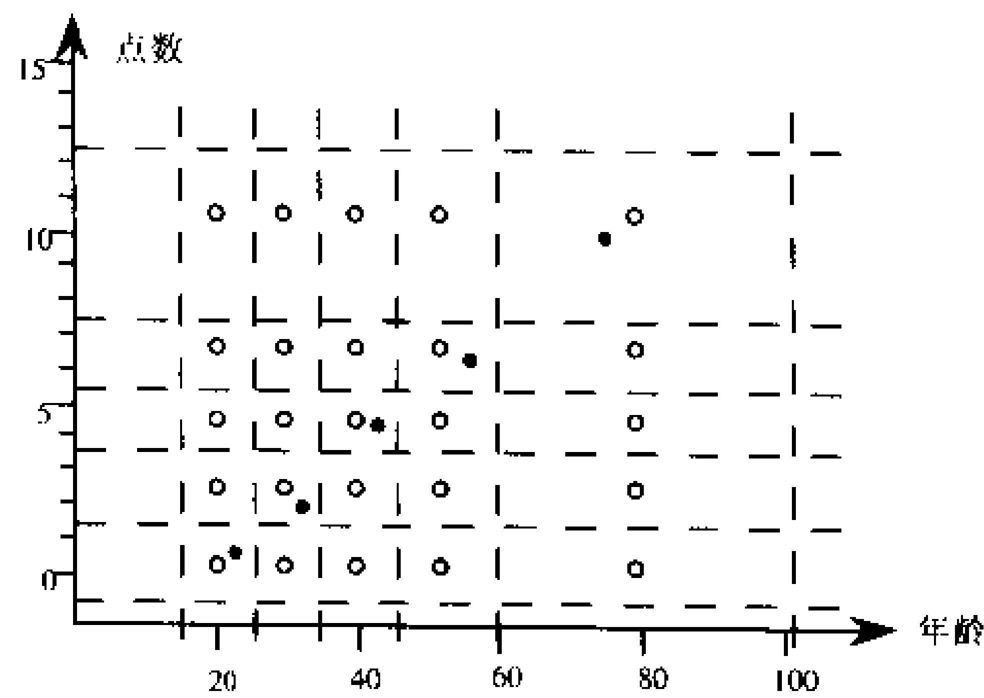


图8-9　保险金计算程序的弱和强等价类测试用例

表8-3　保险金计算程序的决策表测试用例

年龄	16~25	16~25	25~35	25~35	35~45	35~45	45~60	45~60	60~100	60~100
点数	0	1~12	0~2	3~12	0~4	5~12	0~6	7~12	0~4	5~12
年龄系数	2.8	2.8	1.8	1.8	1	1	0.8	0.8	1.5	1.5
安全驾驶折扣	50	-	50	-	100	-	150	-	200	-

决策表测试用例如图8-10所示。

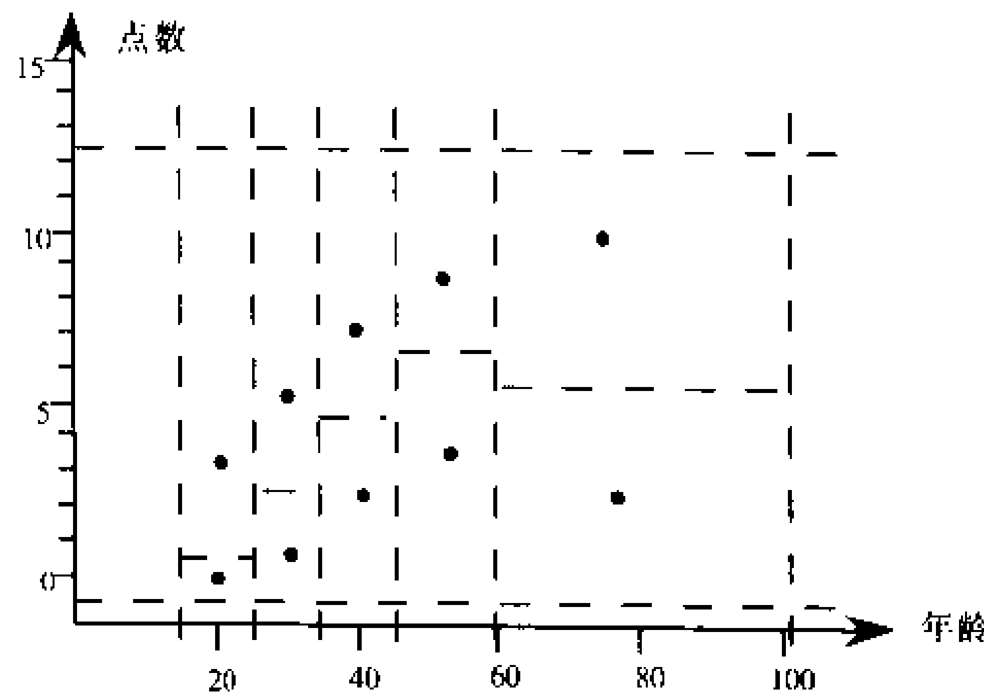


图8-10　保险金计算程序的决策表测试用例

研究一下图8-8和图8-10，一个过多，一个不足。我们希望找到某种折衷，而这正是醉汉

找钥匙故事的意义。保险金计算程序容易在哪些方面出现错误？年龄范围的端点可能是很好的切入点，而这又把我们带回边界值模式。此外，我们没有考虑16岁以下和100岁以上的年龄，这说明要考虑健壮边界值的一些元素。最后，可能还需要检查安全驾驶折扣被取消的值，可能还包括保险不起作用的大于12的点数。（请注意，对这些问题的回答没有出现在问题陈述中，但是测试分析能够启发我们考虑这些问题的答案。）也许这应该叫做混合功能性测试：它利用了通过应用程序的性质决定的所有三种形式测试的优点（图8-11所示的特殊值测试实心圆点）。混合测试看起来很适合，因为这种选择通常可改进测试用例

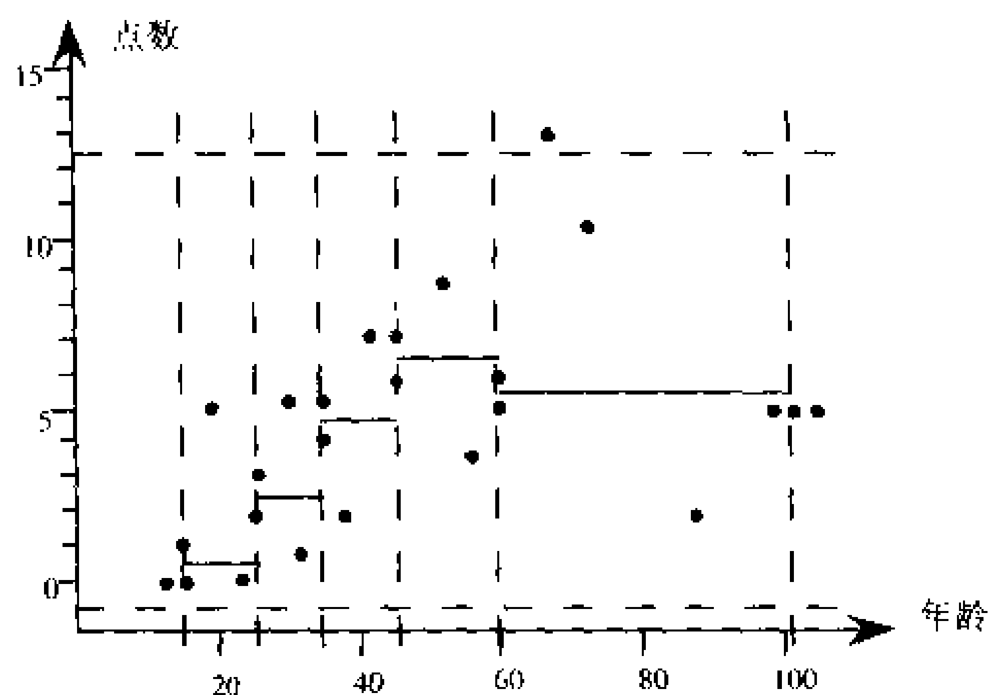


图8-11 保险金计算程序的最终（混合）测试用例

第三部分

结构性测试

第9章
路径测试

结构性测试方法的突出特征，是它们都基于被测程序的源代码，而不是基于定义。由于这种绝对化的基础，因此结构性测试方法支持严格定义、数学分析和精确度量。本章将研究两种最常见形式的路径测试。这些测试方法背后的技术自20世纪70年代就已经提出，这些方法的提出者已经向市场推出实现这些技术的非常成功的工具。这两种技术都从程序图人手，这里再重复一次经过改进的第4章定义。

定义

给定采用命令式程序设计语言编写的一段程序，其程序图是一种有向图，图中的节点表示语句片段，边表示控制流。（完整语句是“默认”的语句片段）。

如果i和j是程序图中的节点，从节点i到节点j存在一条边，当且仅当对应节点j的语句片段可以在对应节点i的语句片段之后立即执行。

根据给定程序构造程序图非常容易，这里采用以伪代码实现的第2章的三角形程序来说明。行号引用语句和语句片段。这里使用一种判断要素：有时将一个片段作为单独节点是很方便的，有时又需要将其与语句的另一部分合并。我们将会看到这些情况都能简化为惟一的DD-路径图，能够消除由于不同判断引人的差别。（数学家会指出，对于给定程序，可以使用多种不同的程序图，所有这些程序图都可以简化为惟一的DD-路径图。）我们还需要判断是否将节点与非可执行语句关联起来，例如变量和类型说明语句。这里我们不做这样的关联。

```
1. Program triangle2 'Structured programming version of simpler specification
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
      'Step 1: Get Input
4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a,b,c)
6. Output("Side A is ",a)
7. Output("Side B is ",b)
8. Output("Side C is ",c)
      'Step 2: Is A Triangle?
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10.     Then IsATriangle = True
11.     Else IsATriangle = False
12  EndIf
      'Step 3: Determine Triangle Type
13. If IsATriangle
14.     Then   If (a = b) AND (b = c)
15.                 Then Output ("Equilateral")
16.                 Else    If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17.                             Then    Output ("Scalene")
```

```
18.                         Else    Output ("Isosceles")
19.                     EndIf
20.             EndIf
21.     Else    Output("Not a Triangle")
22. EndIf
23. End triangle2
```

图9-1给出这段程序的程序图。仔细研究图9-1，可找出第4章讨论过的结构化程序设计构造的图。

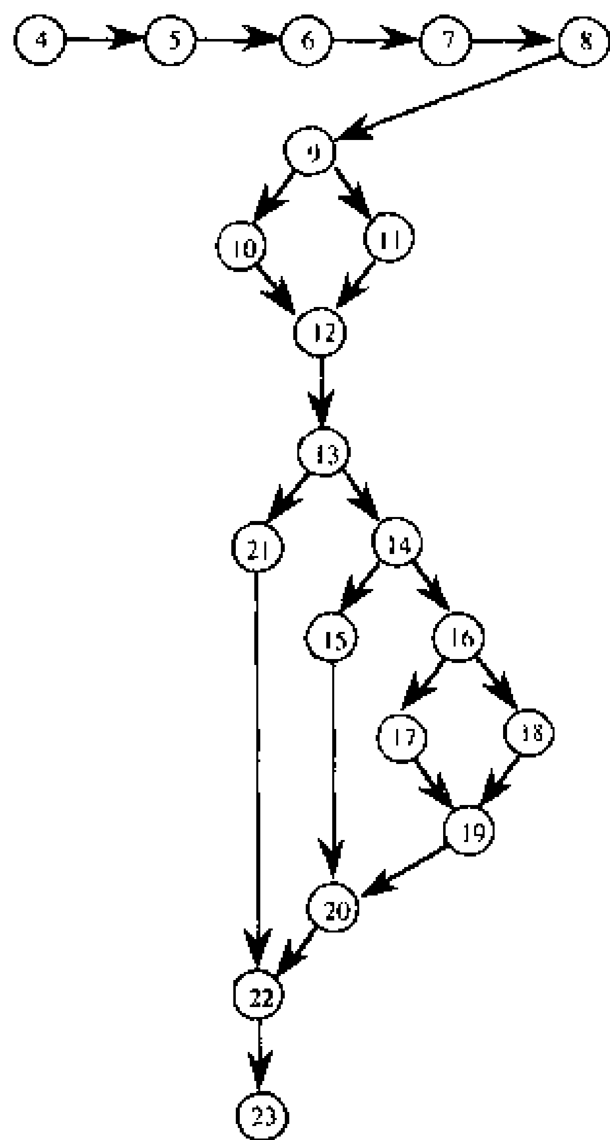


图9-1 三角形程序的程序图

节点4到8是一个序列，节点8到12是一个if-then-else构造，节点15到22是嵌套的if-then-else构造。节点4和23是程序源节点和汇节点，对应单入口、单出口准则。没有循环，因此这是有向不循环图。

程序图的重要性在于，该程序的执行对应于从源节点到汇节点的路径。由于测试用例要完成某条这种程序路径的执行，因此我们有测试用例和测试用例所执行的程序部分之间关系的非常明确的描述。我们还有一种很好的、理论上可预期的方式，处理程序中潜在的大量执行路径。图9-2是一个简单（但未构建的）程序的图，这是用于显示即使是简单程序的完备测试不可能性的一种典型例子（Schach，1993）。在这段程序中，循环范围内从节点B到节点F有五条路径。如果循环最多有18次重复，则存在4.77万亿不同的程序执行路径。

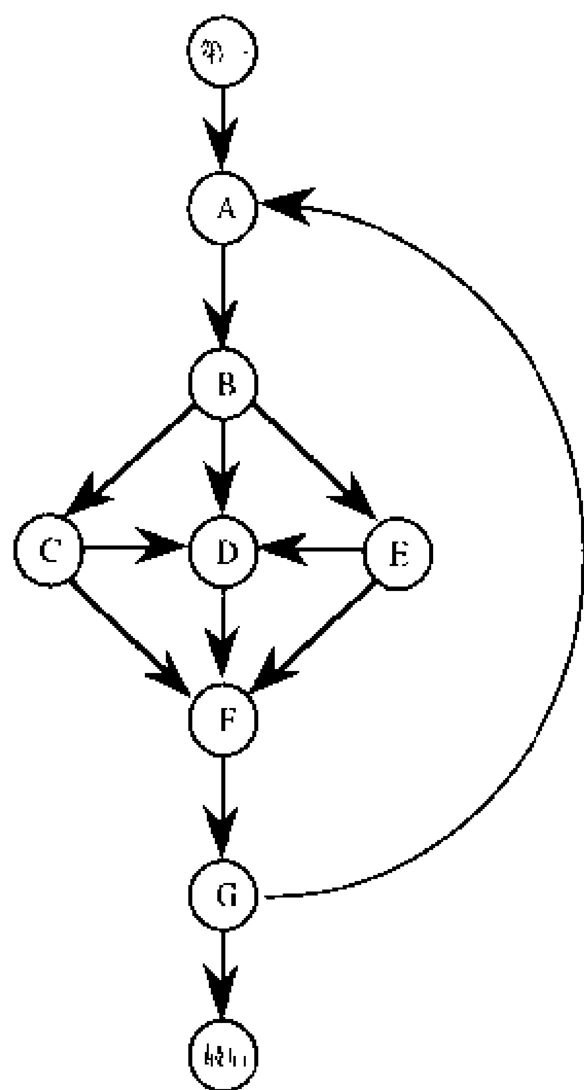


图9-2 数以万亿计的路径

9.1 DD-路径

结构性测试最著名的形式以叫做决策到决策路径（DD-路径）的结构为基础（Miller，1977）。这个名称指语句的一种序列，按照Miller的话说，从决策语句的“出路”开始，到下一个决策语句的“入路”结束。在这种序列中没有内部分支，因此对应的节点像排列起来的一行多米诺骨牌，当第一张牌推倒后，序列中的其他牌也会倒下。Miller的原始定义能够很好地适合第二代语言，例如FORTRAN II，因为决策语句（例如算术IF和DO循环）使用语句标号引用目标语句。对于块结构语言（Pascal、Ada、C），通过语句片段的概念能够解决使用Miller的原始定义所出现的困难，否则会出现有些语句是多个DD-路径成员的程序图。

我们将通过有向图中的节点路径定义DD-路径，可以叫做路径链，其中链是一条起始和终止节点不同的路径，并且每个节点都满足内度 = 1、外度 = 1。请注意，初始节点与链中的所有其他节点2-连接，不会存在1-连接或3-连接，如图9-3所示。图9-3中的链长度（边的数量）是6。有一种长度为0的退化链情况，即链有一个节点和0条边组成。

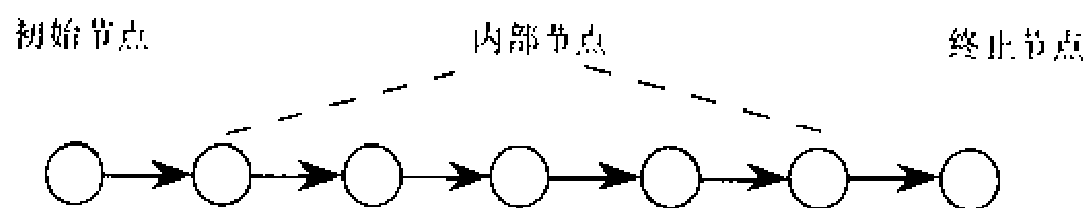


图9-3 有向图中的结点链

定义

DD-路径是程序图中的一条链，使得：

情况1：由一个节点组成，内度＝0
情况2：由一个节点组成，外度＝0
情况3：由一个节点组成，内度≥2或外度≥2
情况4：由一个节点组成，内度＝1并且外度＝1
情况5：长度≥1的最大链

情况1和情况2将结构化程序的程序图的惟一源节点和汇节点建立为初始和最终DD-路径。情况3涉及复杂节点，它保证节点不会包含在多个DD-路径中。情况4用于“短分支”，也用于遵守一个判断、一个DD-路径原则。情况5是“正常情况”，其中DD-路径是单入口、单出口的节点序列（链）。情况5中的“最大”，用于确定正常（非平凡）链的最终节点。

这是一种复杂定义，因此将其用于图9-1所示的程序图。节点4是情况1 DD-路径，我们叫它“第一”。类似地，节点22是情况2 DD-路径，我们叫它“最后”。节点5到8是情况5 DD-路径。我们知道节点8是DD-路径中的最后节点，因为它是遵循链的2-连接性质的最后节点。如果超过节点8包含节点9，就会违反链的内度＝外度＝1准则。如果在节点7处停止，就会违反“最大”准则。节点10、11、15、17和18是情况4 DD-路径。节点9、12、13、14、16、19、20和22是情况3 DD-路径。最后，节点23是情况2 DD-路径。所有这些都归纳在表9-1中。其中，DD-路径的名称对应图9-4中的DD-路径图中的节点名称。

表9-1 图9-1中的DD-路径类型

程序图节点	DD-路径名称	定义情况
4	第一	1
5~8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	最后	2

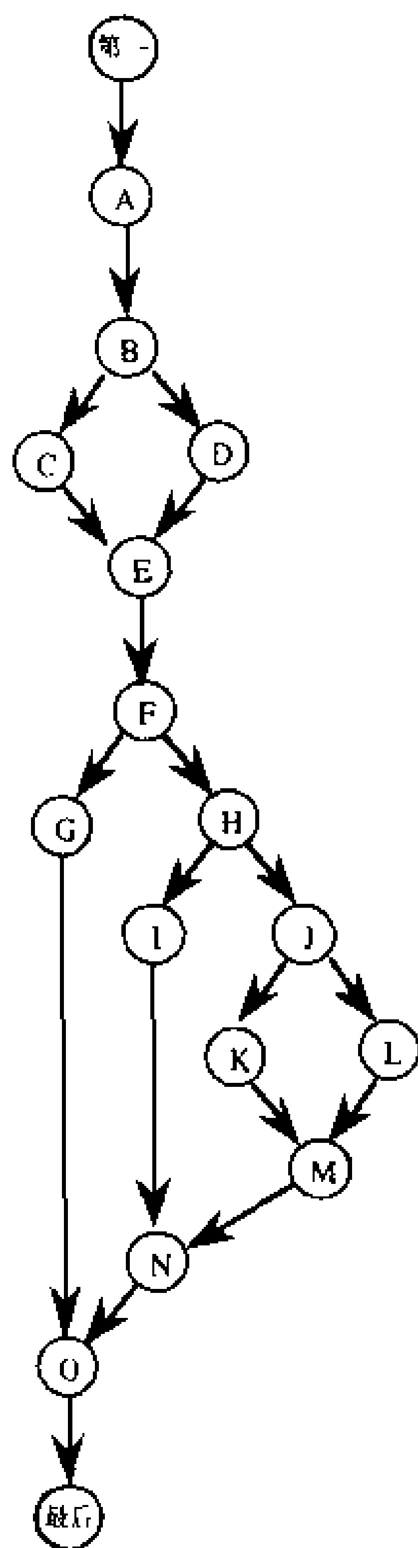


图9-4 三角形程序的DD-路径图

这个例子不够清晰的部分原因是，三角形问题逻辑密集、计算稀疏。这种组合会产生很多很短的DD-路径。如果在计算语句块中包含THEN和ELSE子句，就会有更长的链，就像我们将在佣金问题中看到的那样。下面定义程序的DD-路径图。

定义

给定采用命令式语言编写的一段程序，其DD-路径图是有向图。其中，节点表示其程序图的DD-路径，边表示连续DD-路径之间的控制流。

实际上DD-路径图是一种压缩图（请参阅第4章），在这种压缩图中，2-连接组件被压缩为对应情况5 DD-路径的单个节点。需要这种单节点DD-路径（对应情况1~4）是为了遵循一条语句（或语句片段）恰是一条DD-路径的约定。如果没有这个约定，则会得到很差的DD-路径图，有些语句片段可能会在多个DD-路径中。

这种处理不会使测试人员感到畏惧，因为已经有了能够生成给定程序DD-路径图的高质量商业化工具。厂商可以保证其产品能够广泛适用于各种程序设计语言。在实践中，手工为最多大约100行源代码的程序生成DD-路径图是可行的。如果超过这种规模，大多数测试人员都会需求工具的帮助。

9.2 测试覆盖指标

提出DD-路径的目的，在于DD-路径能够非常精确地描述测试覆盖。第8章曾经提到过，功能性测试的一个基本局限性，就是不能知道通过一组功能性测试用例执行程序对应的冗余的扩展或漏洞的可能性。第1章曾经用一张维恩图表示已描述、已编程和已测试行为。测试覆盖指标，是度量一组测试用例覆盖（或执行）某个程序扩展的工具。

有一些测试覆盖指标被广泛接受并使用，表9-2列出的指标大部分来自E. F. Miller（Miller，1977）的早期研究工作。通过系统地观察程序被测试的范围，能够敏感地管理测试过程。现在的大多数质量机构都把 C_1 指标（DD-路径覆盖）作为测试覆盖的最低可接受级别。语句覆盖指标（C_0）仍然被广泛接受：它是ANSI标准178B强制要求的，并且自从20世纪70年代中期以来，一直在全IBM公司成功地使用。

表9-2 结构性测试覆盖指标

指 标	覆盖描述
C_0	所有语句
C_1	所有DD-路径（判断分支）
C_1p	所有判断的每种分支
C_2	C_1覆盖 + 循环覆盖
C_d	C_1覆盖 + DD-路径的所有依赖对偶
C_{MCC}	多条件覆盖
C_ik	包含最多k次循环的所有程序路径（通常k = 2）
C_{stat}	路径具有"统计重要性"的部分
C_∞	所有可能的执行路径

这些覆盖指标构成一种格（请参阅第10章），有些指标是等价的，有些指标被其他指标蕴涵。格的重要意义在于永远有缺陷类型会在测试的一个层次上被发现，并能够在测试的下层上逃避检测。E. F. Miller发现，当通过一组测试用例满足DD-路径覆盖要求时，可以发现全部缺陷中的大约85%（Miller，1991）。

9.2.1 基于指标的测试

表9-2中的测试覆盖指标告诉我们要测试什么，但是没有说怎么测试。本节将进一步研究根据表9-2中的指标执行源代码的技术。必须清晰地区分：Miller的测试覆盖指标是基于程序图的，其中的节点是完整语句，而我们的表示方式允许语句片段（可以是整个语句）是节点。

语句与判断测试

由于我们的表示方式允许语句片段作为独立的节点，因此语句和判断层次（C_0 和 C_1）可合并为一个问题，在三角形问题（如图9-1所示）中，节点9、10、11和12是一个完整的if-then-else语句。如果要求节点对应完整语句，则可以只执行判断的一个选项并满足语句覆盖准则。由于我们允许语句片段，因此可以很自然地将这种语句分解为三个节点。这样做导致判断分支覆盖。不管是否遵循我们的约定，这些覆盖指标都要求找出一组测试用例，使得当执行时，程序图的所有节点都至少走过一次。

DD-路径测试

如果每条DD-路径都被遍历（C_1指标），则我们知道每个判断分支都被执行，这要求遍历DD-路径图中的每条边。对于if-then和if-then-else语句，这意味着真和假分支都覆盖（C_1p覆盖）。对于CASE语句，每个子句都要覆盖。除此之外，还应该自问为了测试DD-路径，还应该做什么。较长的DD-路径一般代表复杂计算，可以合理地认为是单独的函数。对于这样的DD-路径，应用多个功能性测试可能比较适合，尤其是边界值和特殊值测试。

DD-路径的依赖对偶

C_d涉及第10章将要讨论的问题，即数据流测试。DD-路径对偶之间的最常见的依赖关系是定义/引用关系，其中变量在一个DD-路径中定义（接受值），在另一个DD-路径中引用。这种依赖关系的重要性在于，它们与不可行路径问题有关。我们有很好的DD-路径依赖对偶例子：图9-4中的B和D就是这样的对偶，DD-路径C和L也是。简单DD-路径覆盖可能不会遍历这些依赖关系，因此更深的缺陷类不会被发现。

多条件覆盖

仔细研究DD-路径中的复合条件A和E，不是直接遍历这种判断到其真或假分支，而应该研究可能出现分支的不同方式。一种可能是制做真值表，三个简单条件的复合条件会有八行，产生八个测试用例。另一种可能是将复合判断修改为嵌套的简单if-then-else逻辑，产生更多要覆盖的DD-路径。这里可以看到一种有意思的折衷：语句复杂性和路径复杂性。多条件覆盖可保证这种复杂性不会被DD-路径覆盖所掩盖。

循环覆盖

第4章研究过的压缩图为测试循环问题提供了一种很好的解决方案。人们已经对循环测试做了大量研究，这是有原因的，因为循环是源代码中非常容易出错的部分。在开始讨论之前，先介绍循环很有意思的分类（Beizer，1983）：串联、嵌套和复杂，如图9-5所示。

串联循环是不相交的简单循环序列。嵌套循环是一个循环包含在另一个循环中的循环。如果遵循结构化程序设计规则，就不会出现复杂循环。如果跳转到某个循环内（或跳转出），而这个分支位于其他循环的内部，结果就是Beizer所说的复杂循环。（其他文献定义为棘手循环，这种定义很贴切。）这种循环测试的简单观点是认为每个循环都包含一个判断，并且需要测试判断的两个分支：一个遍历循环，另一个退出（或不进入）循环。这个问题在Huang（1979）中已经做过精心的证明。我们也可以采用经过修改的边界值方法，循环指数按最小值、一般值和最大值给出（请参阅第5章）。我们可以把这种方法进一步推广到全边

界值测试，甚至健壮性测试。如果简单循环的循环体是执行复杂计算的DD-路径，则正如前面已经提到过的，这也应该被测试。一旦测试了循环，就可以将其压缩成一个单独的节点。如果循环是嵌套的，则这个过程从最内层的循环开始，逐步向外重复进行。这会产生我们在边界值分析中发现过的同样的复杂性问题，这是合理的，因为每个循环指数变量都像是输入变量。如果是棘手循环，则必须采用第10章将要讨论的数据流方法仔细分析。作为一种预习，可先考虑如果一个循环修改另一个循环的指数值而可能造成的无限循环。

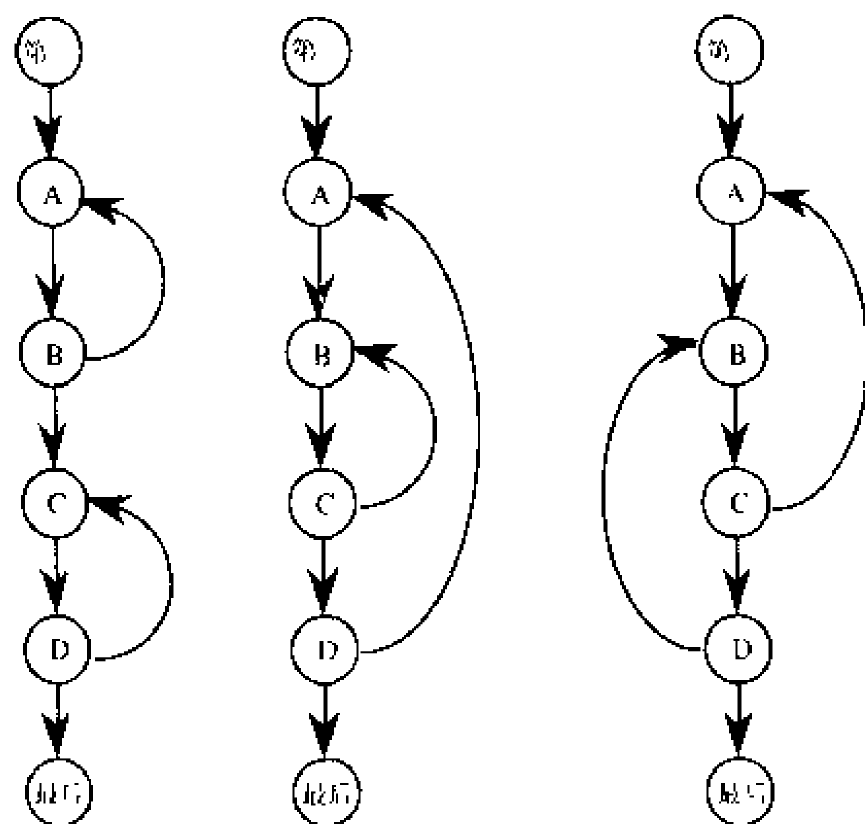


图9-5 串联、嵌套与棘手的循环

9.2.2 测试覆盖分析器

覆盖分析器是一类测试工具，可提供这种方法的自动化测试管理支持。通过覆盖分析器，测试人员可以在经过覆盖分析器“处理”的程序上执行一组测试用例。分析器再使用处理代码所生成的信息生成覆盖报告。例如，对于常见的DD-路径覆盖，处理代码会标识并为原始程序的所有DD-路径加标签。当通过测试用例执行加了处理代码的程序时，分析器会把被每个测试用例遍历的DD-路径制成表格。通过这种方式，测试人员可以实验不同测试用例集合，以确定每个集合的覆盖。

9.3 基路径测试

数学上的“基”的概念对于结构性测试很诱人。特定的集合都可以有一个基，如果有，则基对于整个集合有非常重要的意义。数学家一般采用叫做“向量空间”的结构来定义基，这是元素的一个集合（叫做向量），以及对该向量定义的对应乘法和加法的操作。如果还使

用几个其他准则，则这种结构就是向量空间，所有向量空间都有一个基（事实上可以有多个基）。向量空间的基是相互独立的一组向量，基“覆盖”整个向量空间，使得该空间中的任何其他向量都可以用基向量表示。因此，一组基向量在一定程度上可表示整个向量空间的“本质”：空间中的一切都可以用基表示，并且如果一个基元素被删除，则这种覆盖特性也会丢失。对测试的潜在意义是，如果可以把程序看做是一种向量空间，则这种空间的基就是要测试的非常有意义的元素集合。如果基没有问题，则可以希望能够用基表述的一切都是没有问题的。本节将介绍Thomas McCabe的早期工作，他在20世纪70年代中期就认识到这种可能性。

9.3.1 McCabe的基路径方法

图9-6选自McCabe（1982）。这是一张有向图，可以认为是某个程序的程序图（或DD-路径图）。为了向在其他地方（McCabe，1987；Perry，1987）见到过这个例子的读者提供方便，这里仍然使用节点和边的最初表示法。（请注意，这张图不是来自结构化程序：节点B和C是有两个出口的循环，而且从B到E的边是跳入节点D、E和F中的if-then语句的分支。）这段程序确实拥有单入口（A）和单出口（G）。McCabe将他的测试观点放在图论的主要结果上，说明强连接图的圈数量（请参阅第4章）就是图中线性独立环路的数量。（环路类似于链：不出现内部循环或判断，但是初始节点是终止节点。环路是一组3-连接节点。）

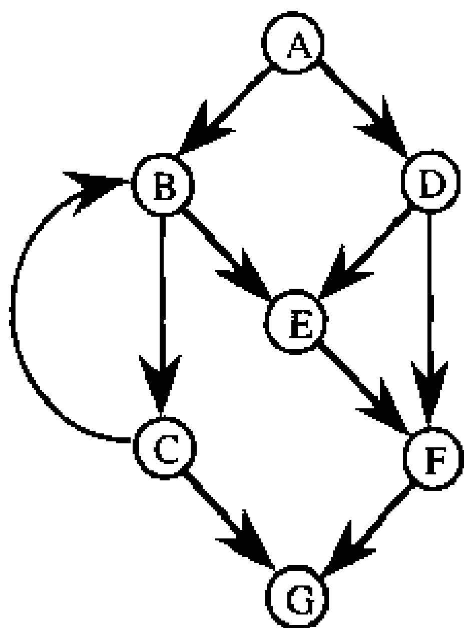


图9-6 McCabe的控制图

通过从（每个）汇节点到（每个）源节点添加一条边，永远都可以创建强连接图。（请注意，如果违反了单入口、单出口规则，会大大增加圈的数量，因为需要从每个汇节点到每个源节点添加边。）图9-7给出了经过这种处理后的结果，图中的边还给出标签，将在后面的讨论中使用。

文献中，在正确表示圈复杂度方面存在一定混乱。有些文献给出的公式是V（G）= e - n + p，有的文献给出的公式是V（G）= e - n + 2p；所有人都认为e是边数，n是节点数，p是

连接区域数。混乱看起来源自任意有向图（例如图9-6给出的有向图），通过从汇节点到源节点添加一条边实现的到强连接有向图的转换（如图9-7所示）。增加一条边显然会影响公式计算出的值，但是不应该影响圈数。以下这种方法可以解决这种不一致问题。在图9-6中，从源节点到吸收节点的线性独立路径数是：

$$V(G) = e - n + 2p$$
$$= 10 - 7 + 2(1) = 5$$

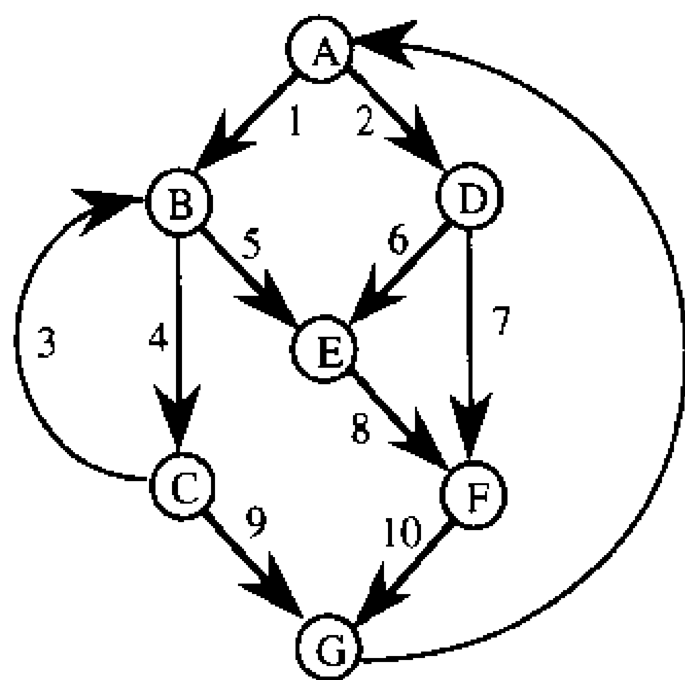


图9-7 McCabe的导出强连接图

图9-7中图的线性独立环路数是：

$$V(G) = e - n + p$$
$$= 11 - 7 + 1 = 5$$

图9-7中的强连接图的圈复杂度是5，因此有五个线性独立环路。如果现在删除从节点G到节点A所添加的边，则这五个环路就成为从节点A到节点G的线性独立路径。在规模很小的框图中，我们可以直观地标识独立路径。以下给出的是用节点序列表示的路径：

p1：A，B，C，G
p2：A，B，C，B，C，G
p3：A，B，E，F，G
p4：A，D，E，F，G
p5：A，D，F，G

通过定义加法和标量乘法的概念，强制使其看起来像向量空间：路径加法就是一条路径后接另一条路径，乘法对应于路径的重复。借助这种公式化，McCabe得到程序路径的向量空间。他对这个框架的基部分的解释是，路径A、B、C、B、E、F、G是基和p2 + p3 − p1，

路径A、B、C、B、C、B、C、G是线性合并2p2 - p1。通过关联矩阵可以更容易地理解这种加法（请参阅第4章），其中，矩阵的行对应路径，列对应边，如表9-3所示。这个表中的条目是按遍历的路径和所经过的边确定的。例如，路径p1经过边1、4和9，路径p2经过边序列1、4、3、4、9。由于边4被路径p2经过了两次，那就是边4列的条目。

表9-3　路径/边的穿越

所经过的路径/边	1	2	3	4	5	6	7	8	9	10
p1：A、B、C、G	1	0	0	1	0	0	0	0	1	0
p2：A、B、C、B、C、G	1	0	**1**	2	0	0	0	0	1	0
p3：A、B、E、F、G	1	0	0	0	**1**	0	0	1	0	1
p4：A、D、E、F、G	0	1	0	0	0	**1**	0	1	0	1
p5：A、D、F、G	0	1	0	0	0	0	**1**	0	0	1
ex1：A、B、C、B、E、F、G	1	0	1	1	1	0	0	1	0	1
ex2：A、B、C、B、C、B、C、G	1	0	2	3	0	0	0	0	1	0

通过观察这个关联矩阵的前五行可检查路径p1 - p5的依赖关系。粗体显示的条目表示只出现在一条路径中的边，因此路径p2 - p5必须是独立的。路径p1独立于所有这些路径，因为任何以其他路径表达p1的企图都会引入不想要的边。没有可以删除的边，这五条路径涉及从节点A到节点G的所有路径集合。这时读者应该检查两个示例路径的线性组合。（对列条目执行加法和乘法运算。）

McCabe接下来开发了一种算法过程（叫做基线方法），用于确定基路径集合。这种方法首先选择一个基线路径，应该对应某个"正常案例"程序执行。这种方法有一定的随意性，McCabe建议选择都有尽可能多的判断节点的路径。接下来重新回溯基线路径，依次"翻转"每个判断点，即当节点的外度≥2时，必须取不同的边。根据McCabe的例子，他先假设通过节点A、B、C、B、E、F、G的路径为基线。（通过前面介绍的p1 - p5路径表示。）这条路径上的第一个判断节点（外度≥2）是节点A，因此对于下一个基路径，要经过边2，而不是边1。得到路径A、D、E、F、G，我们回溯路径1中的节点E、F、G，以便尽可能减少差别。对于下一条路径，可以选第二条路径，取节点D的另一个判断分支，得到路径A、D、F、G。现在只有判断节点B和C没有被翻转，翻转后得到最后两个基路径A、B、E、F、G和A、B、C、G。请注意，这组基路径与表9-3中的基路径集合不同：这没有问题，因为不要求惟一基。

9.3.2　关于McCabe基路径方法的观察

如果读者理解有关基路径和基路径的和、积的讨论有困难，可能会感到怀疑，感到"这可能是对现实世界问题的又一次学术性的过分简化。"这种想法没有错，因为在McCabe的观点中，有两个主要弱点：一是测试基路径集合是充分的（它不是）；第二点与使程序路径看起来像向量空间而进行的类似瑜珈的扭曲有关。McCabe关于路径A、B、C、B、C、B、C、

G是线性组合2p2 – p1的例子非常不能令人满意。2p2部分的含义是什么？要执行路径p2两次吗？（是的，根据所给出的数学公式。）更糟的是，– p1部分的含义是什么？是反向执行路径p1吗？取消对p1的最近一次执行？下次不再执行p1？像这样的数学问题，对于寻求自己的很实际问题解决方案的实践者来说是真正的歧途。为了更好地理解这些问题，我们可以先回到三角形程序例子上来。

首先看图9-4中的三角形程序DD–路径图，先从对应例如不等边三角形的基线路径入手，边是3、4、5。这个测试用例将经过路径p1（请参阅表9-4）。现在，如果翻转节点B处的判断，则得到路径p2。继续这个过程，翻转节点F处的判断，产生路径p3。现在继续翻转基线路径p1上的判断节点，下一个外度 = 2的节点是H。当翻转H时，可得到路径p4。接下来，翻转节点J得到p5。现在我们知道已经完成，因为只有五条基路径，如表9-4所示。

表9-4 图9-4的基路径

原始	p1：A-B-C-E-F-H-J-K-M-N-O-最终	不等边三角形
在B处翻转p1	p2：A-B-D-E-F-H-J-K-M-N-O-最终	不可行
在F处翻转p1	p3：A-B-C-E-F-G-O-最终	不可行
在H处翻转p1	p4：A-B-C-E-F-H-I-N-O-最终	等边三角形
在J处翻转p1	p5：A-B-C-E-F-H-J-L-M-N-O-最终	等腰三角形

现在进行实际检查：如果走过路径p2和p3，会发现两条路径都是不可行的。路径p2不可行，因为通过节点D意味着这些边不构成三角形，因此节点F的判断结果一定是节点G。类似地，在p3中，通过节点C意味着这些边确实会构成三角形，因此节点G不会经过。路径p4和p5都是可行的，分别对应等边三角形和等腰三角形。请注意，非三角形情况没有基路径。

前面已经提到过，输入数据定义域中的依赖关系会给边界值测试造成困难，并且我们通过基于决策表的功能性测试来解决，并在决策表中解决数据依赖性问题。现在我们考虑代码级的依赖关系，这种依赖关系与独立基路径的隐含假设绝对冲突。McCabe的过程成功地标识了在拓扑结构上独立的基路径，但是如果存在矛盾的语义依赖关系，拓扑结构上可行的路径在逻辑上有可能不可行。这个问题的一种解决方案是要求永远翻转语义可行路径中判断结果。另一种方法是找出逻辑依赖性的原因。如果仔细考虑到这个问题，可以找出两条规则：

如果经过节点C，则必须经过节点H。

如果经过节点D，则必须经过节点G。

将这两条规则与McCabe的基线方法结合在一起，可得到以下可行的基路径集合。请注意，如果基路径必须是可行的，则逻辑依赖关系会压缩基路径集合。

p1：A-B-C-E-F-H-J-K-M-N-O-最终	不等边三角形
p6：A-B-D-E-F-G-O-最终	非三角形
p4：A-B-C-E-F-H-I-N-O-最终	等边三角形
P5：A-B-C-E-F-H-J-L-M-N-O-最终	等腰三角形

三角形问题不是很典型，因为没有出现循环。三角形程序只有八条是在拓扑结构上可能的路径，在这八条路径中，只有上面列出的四条基路径是可行的。因此对于这个特例，得到的测试用例与特殊值测试和输出值域测试得到的测试用例相同。

在好的方面，基路径覆盖可保证能够经过所有决策分支，与DD-路径覆盖相同。从基路径关联矩阵描述和三角形程序可行基路径的例子中可以看出这一点。我们可以再前进一步，观察到DD-路径集合的作用与基一样，因为任何程序路径都可以表示为DD-路径的线性组合。

9.3.3　基本复杂度

部分McCabe在圈复杂度上的工作，在改进程序设计方面要比改进测试方面有更大的作用。本节简要综合介绍图论、结构化程序设计及其对测试的影响。这些讨论的核心是基本复杂度概念（McCabe，1982），这是另一种形式压缩图的惟一圈复杂度。前面提到过，压缩图是现有图的一种简化方法，到目前为止，我们的简化都一直限于删除强组件或DD-路径。以下我们围绕图9-8给出的结构化程序设计构造进行压缩。

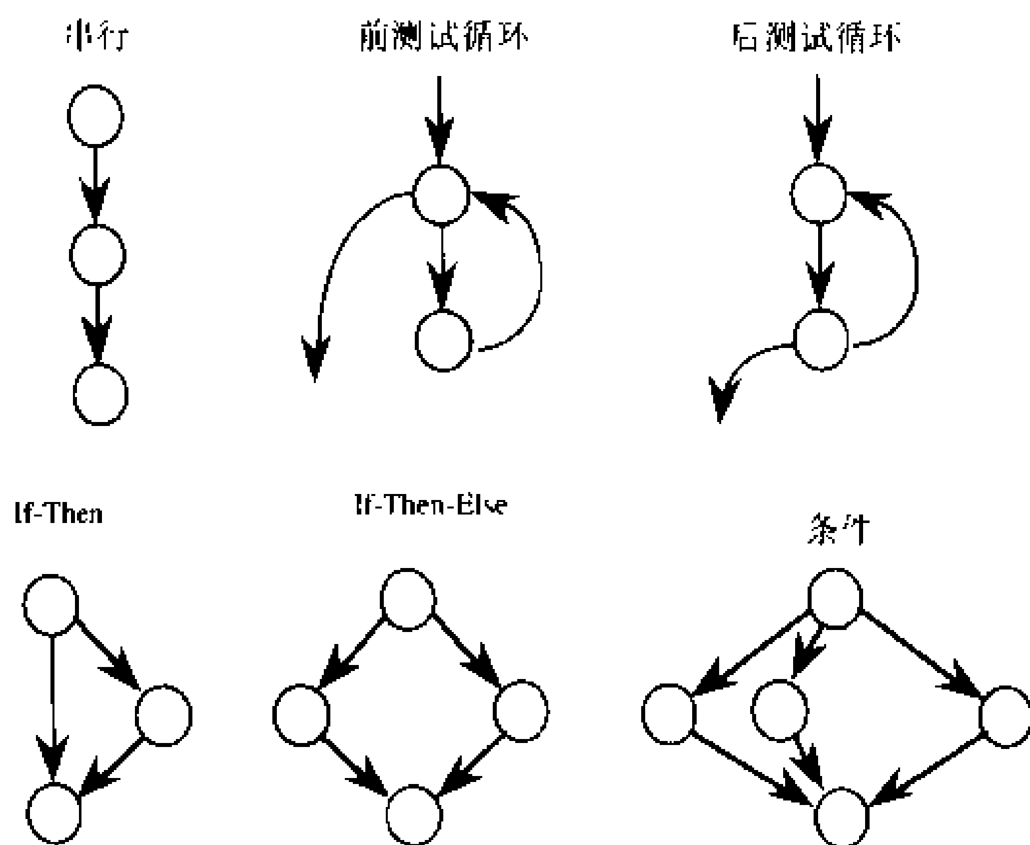


图9-8　结构化程序设计构造

我们的基本思想是寻找一种结构化程序设计构造图，将其压缩成单一的节点，重复这种处理，直到不能再找出其他结构化程序设计构造为止。这个过程可通过由伪代码三角形程序的DD-路径图开始的图9-9和9-10说明，包含节点A、B、C和D的if-then-else构造压缩为节点a，三个if-then构造压缩为节点b、c和d，其余if-then-else（对应于IF IsATriangle语句）构造压缩为节点e，得到圈复杂度V（G）= 1的压缩图。一般来说，如果程序具有很好的结构性（即只包含结构化程序设计构造），则总是可以压缩为只有一条路径的图。

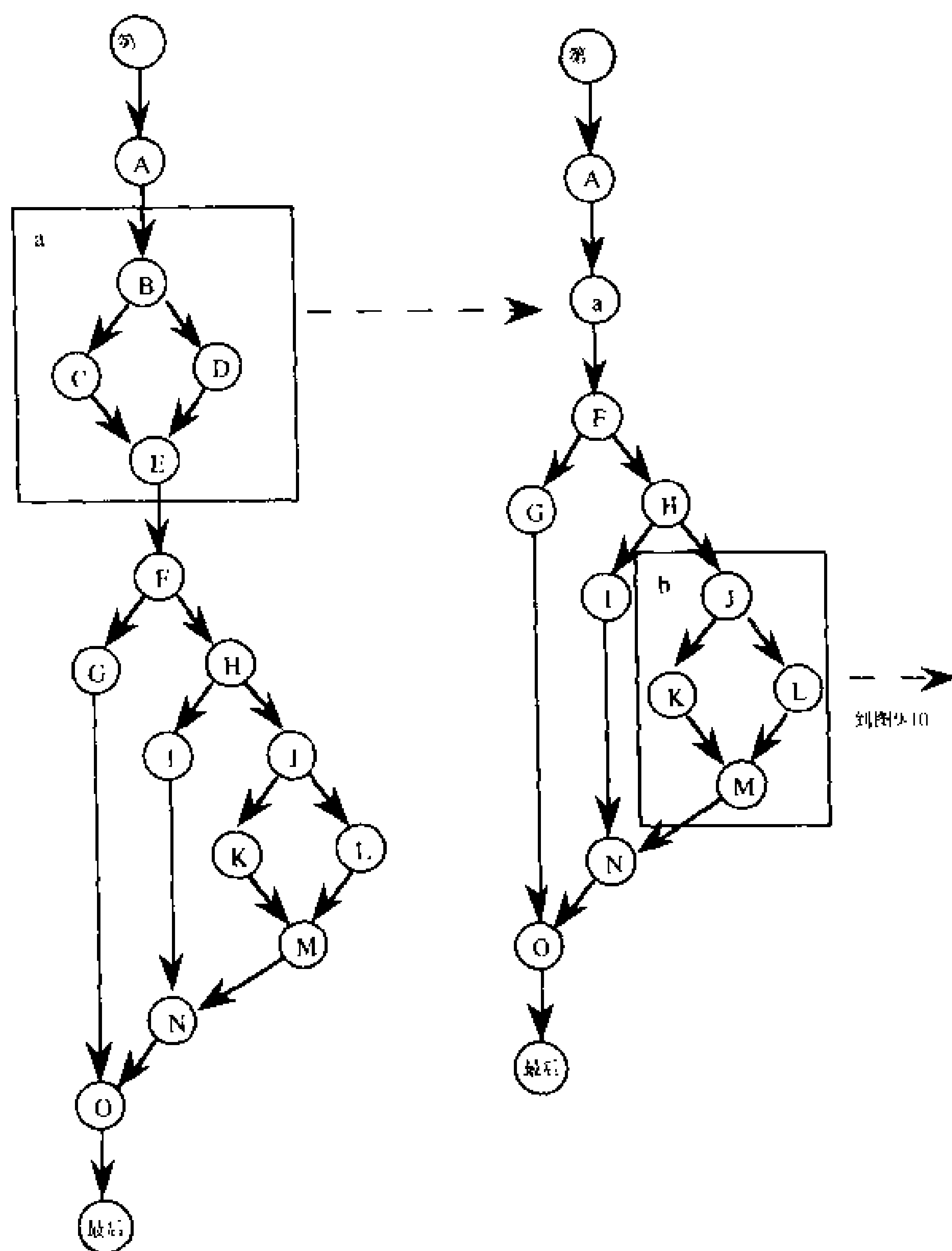


图9-9 根据结构化程序设计构造压缩

图9-6中的图不能采用这种方式压缩（读者可试试看！）。包含节点B和C的循环不能压缩，因为存在从B到E的边。类似地，节点D、E和F看起来像if-then构造，但是从B到E的边与结构要求冲突。McCabe继续寻找与结构化程序设计要求冲突的元素的“非结构”（McCabe，1976），如图9-11所示。

每一个“非结构化”程序设计包含三个不同的路径，与相应的结构化程序设计构造中表示的两个路径相对，所以结论是这样的：冲突增加圈复杂度。McCabe分析中的主要内容是非结构化的程序设计自己不会发生：如果在程序中发生，则至少多于一个，所以程序不会轻易地被非结构化。因为这会引起圈复杂度的增加，测试用例的最小数量也会因此而增加。在下一章中，我们会看到对于数据流测试，非结构化具有有意思的蕴涵。

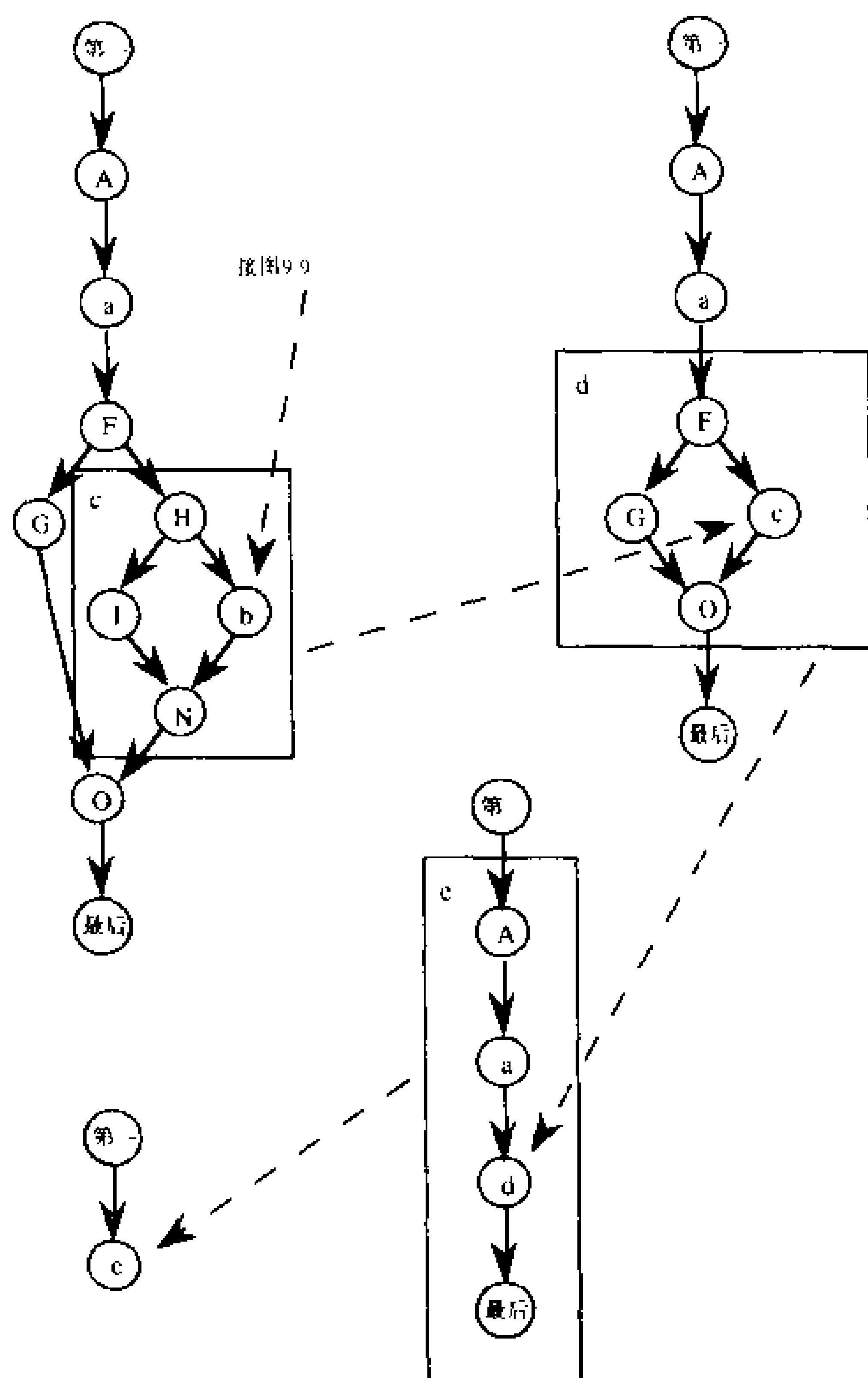


图9-10 根据结构化程序设计构造压缩（续）

测试人员的基本原则是：具有高圈复杂度的程序需要更充分的测试。采用圈复杂度指标的机构，大多数都确定了某种最大可接受复杂度指导方针，一般都选择V（G）＝10。如果单元具有更高的复杂度该怎么办？有两种可能：要么简化单元，要么计划更充分的测试。如果单元的结构很好，则基本复杂度为1，因此可以很容易简化。如果单元的基本复杂度超过了指导方针规定，则最好的选择常常是解决非结构化问题。

9.4 指导方针与观察

我们在研究功能性测试过程中，观察到漏洞和冗余都存在，而且同时不能被发现。原因

是功能性测试使我们离代码过远。结构性测试的路径测试方法又在另一个方向上走得太远：将代码采用有向图表示和程序路径公式化，掩盖了代码中的重要信息，具体地说就是可行路径和不可行路径的区别。下一章将研究基于数据流的测试。这些技术更接近代码，因此从路径分析这种极端方向上返回来。

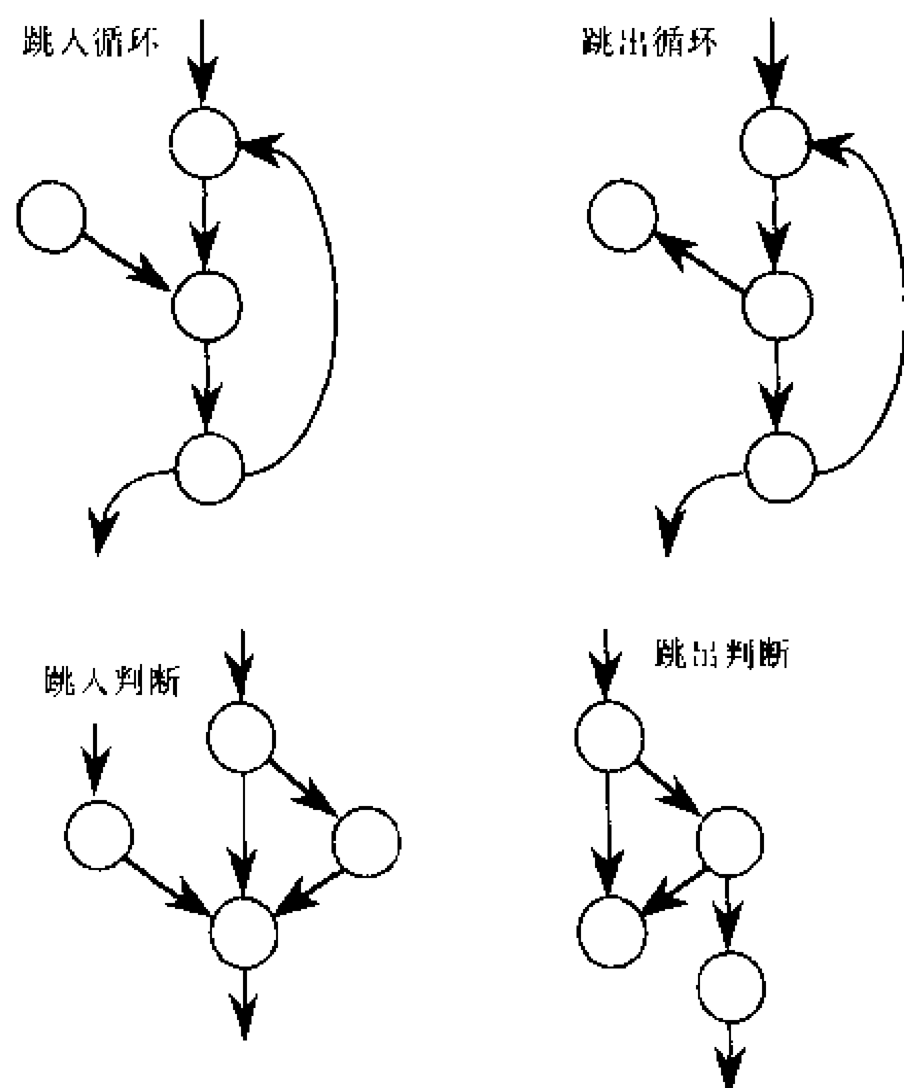


图9-11 与结构化程序设计的冲突

McCabe的论点部分正确。"重要的是理解这些纯粹是度量测试质量的指标，而不是标识测试用例的过程"（McCabe，1982）。他指的是DD-路径覆盖指标（等价于判断分支指标）和圈复杂度指标（要求经过不同程序路径的最低圈数），因此，基路径测试给出了必须进行的测试的下限。

基于路径的测试还提供了用做功能性测试交叉检查的一组指标。可以使用这些指标解决漏洞和冗余问题。如果发现同一条程序路径被多个功能性测试性用例遍历，就可以怀疑这种冗余不会发现新的缺陷。如果没有达到一定的DD-路径覆盖，则可以知道在功能性测试用例中存在漏洞。例如，假设有一个程序包含大量错误处理，我们采用边界值测试用例（最小值、略大于最小值、正常值、略小于最大值和最大值）。由于这些都是允许值，所以DD-路径对应于不会经过的错误处理代码。如果增加根据健壮性测试或传统等价类测试导出的测试用例，则DD-路径就会得到提高。超出覆盖指标的这种明显使用方式，存在着真正测试工艺师的机会。表9-2列出的覆盖指标可以有两种使用方式：作为一种强制执行的标准（例如所有单元都要达到全DD-路径覆盖测试），或作为一种机制，以便指导对比其他代码要求更严格的部

分代码有选择性地进行测试，对于具有复杂逻辑的模块可能选择多条件覆盖，而对于大量迭代处理的模块则采用循环覆盖指标指导测试。这也许是一种最好的结构性测试观点：利用源代码的性质标识合适的覆盖指标，然后再使用这些指标交叉检查功能性测试用例。如果所要求的覆盖没有达到，则根据有意义的路径标识额外的（特殊值）测试用例。

现在应该再回头看看第1章介绍过的测试维恩图。图9-12给出了已描述行为（集合S）、已编程行为（集合P）和程序中在拓扑结构上可行的路径（集合T）。通常区域1是最需要的，因为它包含由可行路径实现的已描述行为。根据定义，所有可行路径在拓扑结构上都是可行的，因此集合P的阴影部分（区域2和6）必须为空。区域3包含对应未描述行为的可行路径，需要检查这种额外功能：如果这些功能有用，则应该修改规格说明，否则应该删除这些可行路径。区域4和7包含不可行路径，其中区域4是有疑问的。区域4指几乎被实现了的已描述行为，即拓扑结构可能但是不可行的程序路径。这种区域非常可能对应代码错误，需要通过修改使路径可行。区域5仍然对应没有实现的已描述行为。基于路径的测试永远也不会发现这种区域。最后，区域7很奇怪：未定义、不可行，但是从拓扑结构看是可能的路径。严格地说，这里不会出现问题，因为不会执行不可行路径。如果对应的代码被维护人员不正确地修改（也可能是没有充分理解该代码的程序员），这些路径可能变为可行路径，与区域3一样。

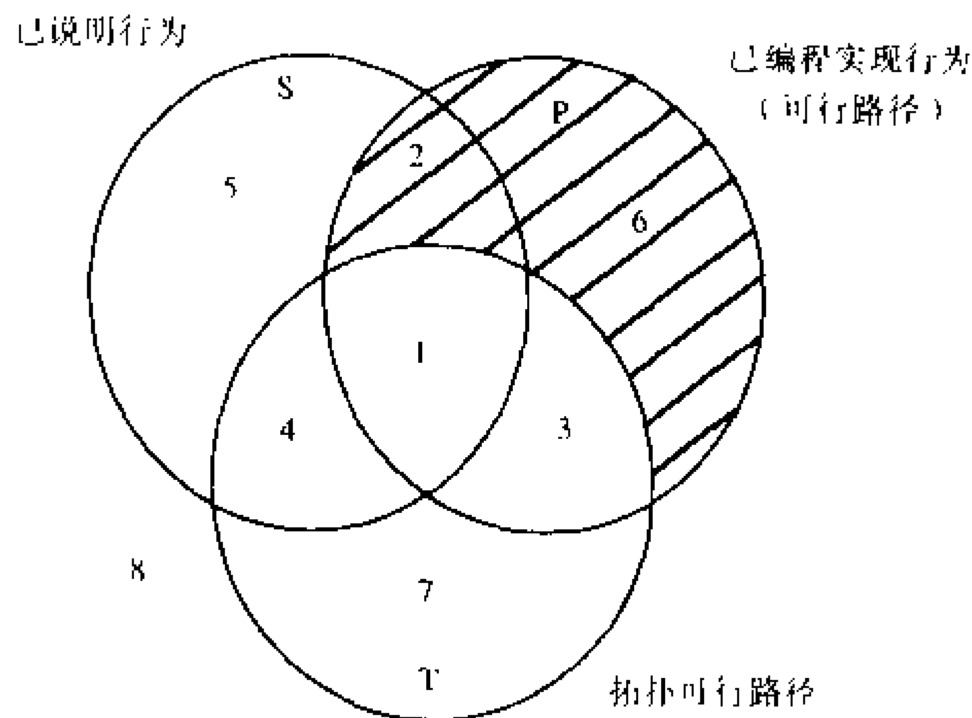


图9-12　可行的和在拓扑结构上可能的路径

9.5　参考文献

Huang, J.C., Detection of dataflow anomaly through program instrumentation, *IEEE Transactions on Software Engineering*, SE-5; pp. 226–236, 1979.

Schach, Stephen R., *Software Engineering*, 2nd ed., Richard D. Irwin, Inc., and Aksen Associates, Inc., 1993.

Miller, E.F., *Tutorial: Program Testing Techniques,* COMPSAC '77 IEEE Computer Society, 1977

Miller, Edward F. Jr., Automated software testing: a technical perspective, *Amer Programmer,* Vol. 4, No. 4, April 1991, pp. 38–43.

McCabe, Thomas J., A complexity metric, *IEEE Transactions on Software Engineering,* SE-2, 4, December 1976, pp. 308–320.

McCabe, Thomas J., Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, National Bureau of Standards (Now NIST), Special Publication 500-99, Washington, D.C., 1982.

McCabe, Thomas J., *Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric,* McCabe and Associates, Baltimore, 1987.

Perry, William E., *A Structured Approach to Systems Testing,* QED Information Systems, Inc., Wellesley, MA, 1987.

9.6 练习

1. 请给出图9-2中图的圈复杂度。

2. 请标识图9-2中图的一个基路径集合。

3. 请讨论McCabe外度≥3节点的“翻转”概念。

4. 假设以图9-2作为某段程序的DD-路径图。请为C_0、C_1和C_2指标开发（可以是测试用例的）路径集合。

5. 请为伪代码三角形程序开发多条件覆盖测试用例。（请注意语句片段14和16之间表达式（a = b）AND（b = c）的依赖关系。）

6. 请重新编写程序片段14~20，用嵌套if-then-else语句替代复合条件。请比较你的程序和现有版本的圈复杂度。

```
14. If (a = b) AND (b = c)
15.      Then Output ("等边三角形")
16.      Else If (a≠b) AND (a≠c) AND (b≠c)
17.           Then Output ("不等边三角形")
18.           Else Output ("等腰三角形")
19.      EndIf
20. EndIf
```

7. 请仔细研究原始语句片段14~20。对于a = c测试用例（例如a = 3、b = 4、c = 3）会出现什么情况？第14行中的条件利用等式的传递性去掉了a = c条件。这会产生问题吗？

8. CRC Web站点上的whiteBox.exe出现包含三角形、NextDate和佣金问题的Visual Basic实现。输出显示的是测试用例（集合）的DD-路径覆盖。请使用这个程序执行各种测试用例集合，以确定各种功能性测试技术的DD-路径覆盖。

9. （只供数学家。）为了使集合V成为某个向量空间，必须为该集合中的元素定义两种

操作（加法和标量乘法）。在加法中，对一切向量x、y、z ∈ V，以及一切标量k、l、0和1，必须遵守以下准则：

a. 如果x、y ∈ V，则向量x + y ∈ V。

b. x + y = y + x。

c. （x + y） + z = x + （y + z）。

d. 存在一个向量0 ∈ V，使得x + 0 = x。

e. 对于任何x ∈ V，存在一个向量 − x ∈ V，使得x + （ − x） = 0。

f. 对于任何x ∈ V，向量kx ∈ V。

g. k（x + y） = kx + ky。

h. （k + l）x = kx + lx。

i. k（lx） = （kl）x。

j. 1x = x。

程序中的路径“向量空间”遵守这10条准则中的多少条？

第10章

数据流测试

数据流测试是一个不大合适的术语，因为它容易使人联想到数据流图，实际上两者并没有什么联系。数据流测试指关注变量接收值的点和使用（或引用）这些值的点的结构性测试形式。数据流测试用做路径测试的“真实性检查”。实际上，很多数据流测试支持者（和研究人员）也都的确把这种方法看做是一种路径测试。本章将介绍两种主要形式的数据流测试：一种提供一组基本定义和一种统一的测试覆盖指标结构，另一种基于叫做“程序片”的概念。这两种方法都形式化了测试人员的直觉行为（和分析），尽管这些方法都从程序图入手，但是都向功能性测试用例靠拢。

大多数程序都通过数据交付功能。表达数据的变量要接收值，并被用来为其他变量计算值。自从20世纪60年代初以来，程序员一直通过变量接收值的点（语句）和使用这些值的点分析源代码。很多时候，这种分析以列出变量名出现的语句行号的索引表为基础进行。索引表是第二代语言编译器的常见功能（COBOL程序员仍然经常使用）。早期的数据流分析常常集中于现在叫做定义/引用异常的缺陷：

变量被定义，但是从来没有使用（引用）。
所使用的变量没有被定义。
变量在使用之前被定义两次。

这些异常可以通过程序的索引表发现。由于索引表信息是由编译器生成的，因此这些异常可以通过所谓静态分析发现，即在不执行被测程序的情况下发现源代码失效。

10.1 定义/使用测试

定义/使用测试的大部分形式化工作是在20世纪80年代初完成的（Rapps，1985），本节给出的定义都与Clarke（1989）给出的定义兼容，Clarke（1989）归纳了大部分定义/使用测试理论。这部分研究与第4章和第9章中的公式非常一致，假设程序图中的节点代表语句片段（语句片段可以是整个语句），并且程序遵循结构化程序设计规则。

以下定义引用拥有程序图G（P）的程序P和一组程序变量V。程序图G（P）按第4章介绍的方式构造，语句片段代表节点，边代表节点序列。G（P）有一个单入口节点和一个单出口节点，并且不允许有从某个节点到其自身的边。路径、子路径和环路都与第4章的定义相同。P中的所有路径集合是PATHS（P）。

定义

节点n ∈ G（P）是变量v ∈ V的定义节点，记做DEF（v，n），当且仅当变量v的值由对应节点n的语句片段处定义。

输入语句、赋值语句、循环控制语句和过程调用，都是定义节点语句的例子。如果执行对应这种语句的节点，那么与该变量关联的存储单元的内容就会改变。

定义

节点n ∈ G（P）是变量v ∈ V的使用节点，记做USE（v，n），当且仅当变量v的值在对应节点n的语句片段处使用。

输出语句、赋值语句、条件语句、循环控制语句和过程调用，都是使用节点语句的例子。如果执行对应这种语句的节点，那么与该变量关联的存储单元的内容会保持不变。

定义

使用节点USE（v，n）是一个谓词使用（记做P-use），当且仅当语句n是谓词语句；否则，USE（v，n）是计算使用（记做C-use）。

对应于谓词使用的节点永远有外度≥2，对应于计算使用的节点永远有外度≤1。

定义

关于变量v的定义-使用路径（记做du-path）是PATHS（P）中的路径，使得对某个v ∈ V，存在定义和使用节点DEF（v，m）和USE（v，n），使得m和n是该路径的最初和最终节点。

定义

关于变量v的定义清除路径（记做dc-path），是具有最初和最终节点DEF（v，m）和USE（v，n）的PATHS（P）中的路径，使得该路径中没有其他节点是v的定义节点。

测试人员应该注意到这些定义如何用所存储数据值捕获的计算的关键。定义-使用路径和定义清除路径描述了跨从值被定义的点到值被使用的点的源语句的数据流。不是定义-清除的定义-使用路径，是潜在有问题的地方。

10.1.1　举例

以下仍然使用佣金问题及其程序图说明这些定义。下面给出有编号的伪代码，后面给出的是根据第4章所讨论的过程构建的程序图。这个程序根据所销售的枪机、枪托和枪管总数确定的销售额来计算佣金。其中的while循环是经典的哨兵控制循环，当枪机值为－1时，说明销售数据结束。总销售量通过在while循环中累加数据值得到。打印出这种初步信息之后，利用程序开头定义的商品价格常数来计算销售额。然后，销售额被程序的条件部分用来计算佣金。

```
Program Commission (INPUT,OUTPUT)

    Dim locks, stocks, barrels As Integer
    Dim lockPrice, stockPrice, barrelPrice As Real
    Dim totalLocks, totalStocks, totalBarrels As Integer
    Dim lockSales, stockSales, barrelSales As Real
    Dim sales, commission As Real
```

```
7    lockPrice = 45.0
8    stockPrice = 30.0
9    barrelPrice = 25.0
10   totalLocks = 0
11   totalStocks = 0
12   totalBarrels = 0

13   Input(locks)
14   While NOT(locks = -1)    'loop condition uses -1 to indicate end of data
15       Input(stocks, barrels)
16       totalLocks = totalLocks + locks
17       totalStocks = totalStocks + stocks
18       totalBarrels = totalBarrels + barrels
19       Input(locks)
20   EndWhile

21   Output("Locks sold: ", totalLocks)
22   Output("Stocks sold: ", totalStocks)
23   Output("Barrels sold: ", totalBarrels)
24   lockSales = lockPrice*totalLocks
25   stockSales = stockPrice*totalStocks
26   barrelSales = barrelPrice * totalBarrels
27   sales = lockSales + stockSales + barrelSales
28   Output("Total sales: ", sales)

29   If (sales > 1800.0)
30       Then
31           commission = 0.10 * 1000.0
32           commission = commission + 0.15 * 800.0
33       commission = commission + 0.20*(sales-1800.0)
34   Else If (sales > 1000.0)
35       Then
36                   commission = 0.10 * 1000.0
37                   commission = commission + 0.15*(sales-1000.0)
38           Else commission = 0.10 * sales
39           EndIf
40   EndIf

41   Output("Commission is $", commission)
42   End Commission
```

图10-2给出的是图10-1程序图的决策到决策路径（DD-路径）图。在这个DD-路径图中做了更多压缩，因为佣金问题中有很多计算。表10-1给出了与DD-路径关联的语句片段。

表10-1　图10-1中的DD-路径

DD-路径	节　点
A	7，8，9，10，11，12，13
B	14
C	15、16、17、18、19
D	20、21、22、23、24、25、26、27、28
E	29
F	30，31，32，33

（续）

DD-路径	节　点
G	34
H	35，36，37
I	38
J	39
K	40
L	41，42

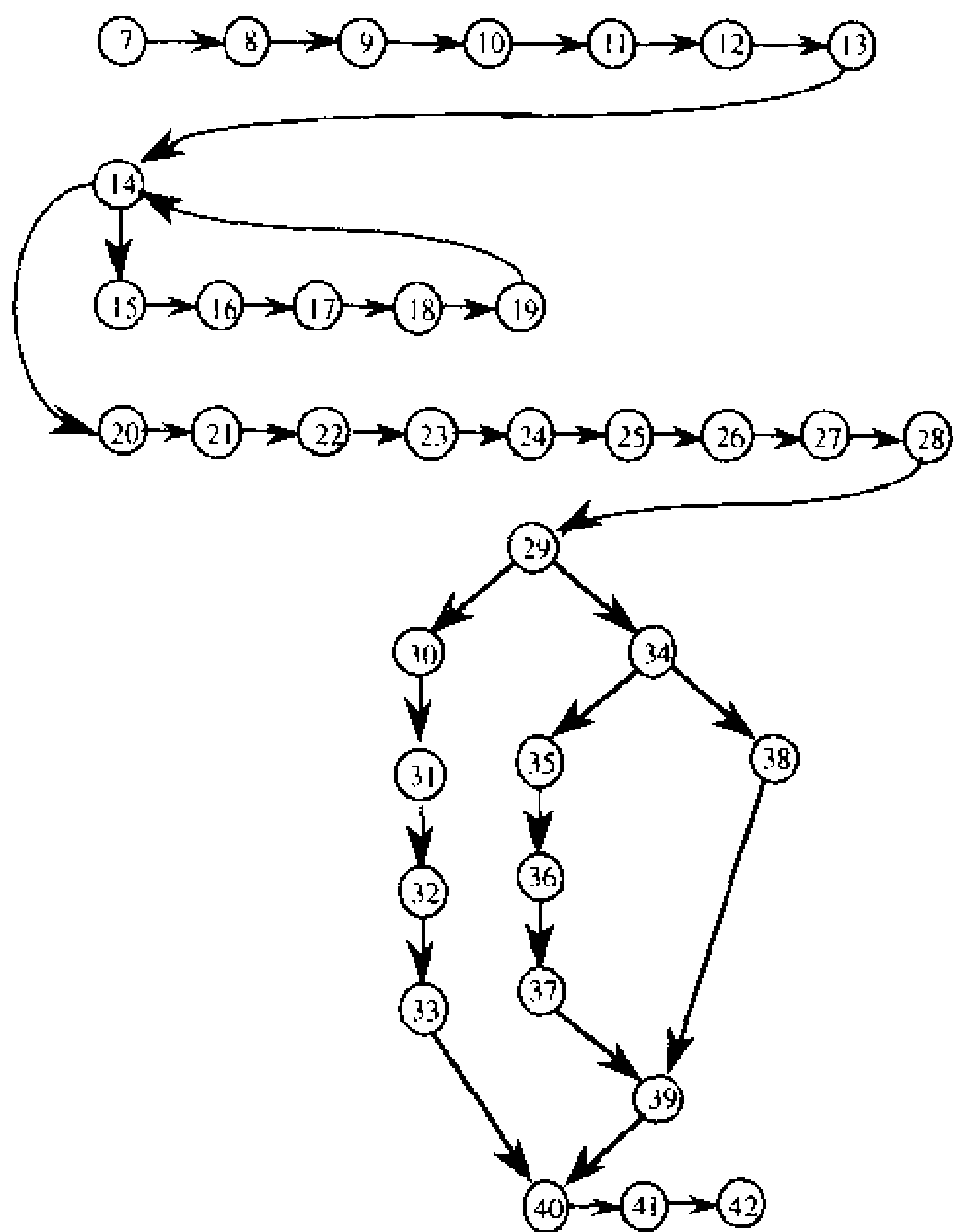


图10-1　佣金问题的程序图

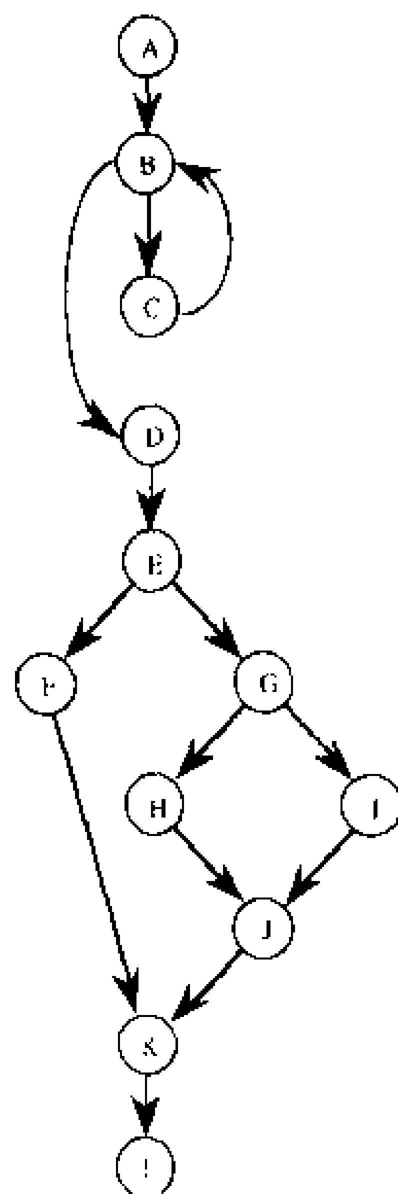


图10-2　佣金问题的DD-路径图

表10-2列出了佣金问题变量的定义节点和使用节点。我们使用这些信息，结合图10-1中的程序图，标识各种定义-使用和定义清除路径。究竟非可执行语句，例如常量和变量说明语句是否应该被认为是定义节点，是学术界争论的问题。如果沿定义-使用路径跟踪执行情况，则这种节点并不非常重要。但是如果出现错误，则包含这种节点会有帮助。可自己做

出选择。我们将各种路径表示为节点编号序列。

表10-2 佣金问题变量的定义/使用节点

变 量	定义节点	使用节点
lockPrice	7	24
stockPrice	8	25
barrelPrice	9	26
totalLocks	10，16	16，21，24
totalStocks	11，17	17，22，25
totalBarrels	12，18	18，23，26
locks	13，19	14，16
stocks	15	17
barrels	15	18
lockSales	24	27
stockSales	25	27
barrelSales	26	27
sales	27	28，29，33，34，37，38
commission	31，32，33，36，37，38	32，33，37，41

表10-3给出了佣金问题中的部分定义-使用路径，采用（图10-1中的）开始和结束节点命名。表10-3的第三列表示定义-使用路径是否是定义清除的。有些定义-使用路径是很简单的，例如lockPrice、stockPrice和barrelPrice。有些定义-使用路径更复杂一些：while循环（节点序列<14，15，16，17，18，19，20>）输入和totalLocks、totalStocks和totalBarrels的累加值。表10-3只显示了totalStocks变量的细节。TotalStocks的初始值定义出现在节点11上，在节点17上第一次使用。因此，由节点序列<11，12，13，14，15，16，17>组成的路径（11，17）是定义清除的。由节点序列<11，12，13，（14，15，16，17，18，19，20）*，21，22>组成的路径（11，22）不是定义清除的，因为totalStocks在节点11和17（可能在节点17处多次定义）处定义。（while循环后面的星号是Kleene星表示法，用于形式逻辑和一般表达式，表示零或更多次重复。）

表10-3 部分定义/使用路径

变 量	路径（开始，结束）节点	是定义清除吗？
lockPrice	7，24	是
stockPrice	8，25	是
barrelPrice	9，26	是
totalStocks	11，17	是
totalStocks	11，22	否
totalStocks	17，25	否
totalStocks	17，17	是
totalStocks	17，22	否
totalStocks	17，25	否

（续）

变　量	路径（开始，结束）节点	是定义清除吗？
locks	13，14	是
locks	19，14	是
locks	13，16	是
locks	19，16	是
sales	27，28	是
sales	27，29	是
sales	27，33	是
sales	27，34	是
sales	27，37	是
sales	27，38	是

10.1.2　stocks的定义-使用路径

首先研究一条简单路径：变量stocks的定义-使用路径。我们有DEF（stocks，15）和USE（stocks，17），因此路径<15，17>是一个关于stocks的定义-使用路径。stocks没有其他定义节点，因此这条路径是定义清除路径。

10.1.3　locks的定义-使用路径

两个定义节点和两个使用节点是locks变量更有意思：我们有DEF（locks，13）、DEF（locks，19），USE（locks，14）和USE（locks，16），产生四条定义-使用路径：

p1 = <13，14>

p2 = <13，14，15，16>

p3 = <19，20，14>

p4 = <19，20，14，15，16>

定义-使用路径p1和p2引用了locks的填充值，在节点13读入；locks在while语句（节点14）中有一个谓词使用，如果这个条件为真（例如路径p2），则在语句16处有一个计算使用。其他两个定义-使用路径在while循环接近结尾处，在循环重复时出现。如果“扩展”路径p1和p3以包含节点21

p1' = <13，14，21>

p3' = <19，20，14，21>

则p1'、p2、p3'和p4构成while循环测试用例的非常完备的集合——旁路循环、开始循环、重复循环和退出循环。所有这些定义-使用路径都是定义清除路径。

10.1.4　totalLocks的定义-使用路径

totalLocks的定义-使用路径会产生典型的计算测试用例。通过两个定义节点（DEF（totalLocks，10）和DEF（totalLocks，16））和三个使用节点（USE（totalLocks，16）、USE（totalLocks，21）和USE（totalLocks，24）），预期应该得到六条定义-使用路径。以下再深入研究一下。

> 路径p5 = <10，11，12，13，14，15，16>是一条定义-使用路径，其中的totalLocks（0）初始值有一个计算使用。这条路径是定义清除的。下一条路径有问题：

p6 = <10，11，12，13，14，15，16，17，18，19，20，21>

路径p6忽略了while循环重复的可能性。子路径<16，17，18，19，20，14，15>可以多次经过能够说明这一点，暂且先不讨论这一点。还有一条定义-使用路径不是定义清除的。如果节点21处（Output语句）的totalLocks值出现问题，就要考虑DEF（totalLocks，16）节点的介入。

下一条路径包含p6，可以通过在其对应的节点序列处使用路径名称表示：

p7 = <10，11，12，13，14，15，16，17，18，19，20，21，22，23，24>

p7 = <p6，22，23，24>

定义-使用路径p7不是定义清除的，因为它包含节点16。

以节点16（赋值语句）开始的子路径很有意思。首先，<16，16>是退化。如果将其“扩展”到机器码，就能够将定义和使用部分分开。我们不允许这种情况作为定义-使用路径。在技术上，赋值语句的右侧引用节点11定义的值（请参阅路径p5）。其他两条定义-使用路径都是p7的子路径：

p8 = <16，17，18，19，20，21>

p9 = <16，17，18，19，20，21，22，23，24>

这两条定义-使用路径都是定义清除的，并且都有前面讨论过的循环迭代问题。

10.1.5　sales的定义-使用路径

sales只使用了一个定义节点，因此关于sales的所有定义-使用路径都必须是定义清除的。我们对这些定义-使用路径感兴趣，是因为这些路径可说明谓词和计算使用。头三条定义-使用路径很容易：

p10 = <27，28>

p11 = <27，28，29>

p12 = <27，28，29，30，31，32，33>

请注意，p12是一个有三个使用节点的定义清除路径，它还包含路径p10和p11。如果要测试p12，就会知道测试还会覆盖其他两条路径。本章结束时还会讨论这个问题。

语句29~40的IF、ELSE IF逻辑显示出最初研究中的一种任意性。以路径p11开始的定义-使用路径有两种选择：静态选择是路径<27，28，29，30，31，32，33，34>，动态选择是路径<27，28，29，34>。这里我们使用动态选择，因此sales的剩余定义-使用路径是：

p13 = <27，28，29，34>

p14 = <27，28，29，34，35，36，37>

p15 = <27，28，29，38>

请注意，动态选择与第9章中的DD-路径使用考虑非常一致。

10.1.6　commission的定义-使用路径

如果读者仔细思考上面的讨论，可能会对关于commission的定义-使用路径讨论感到畏惧，这种担心是对的——现在该变化分析方式了。语句29~41中，佣金的计算由变量sales的范围控制。语句31~33使用存储单元保存中间值，累加佣金值。这是一种常见程序设计实践，给出这种处理是为了说明最终值是怎样计算的。（我们可以用语句"commission：= 220 + 0.20*（sales-1800）"取代那些程序。其中，220是通过0.10*1000 + 0.15*800计算得到的，但是维护人员很难理解。）这种"累加"版本使用中间值，在定义-使用路径分析中表现为定义节点和使用节点。我们决定将像31和32这样的赋值语句从定义-使用路径中排除，所以只考虑从三个"实际"定义节点开始的定义-使用路径：DEF（commission，33）、DEF（commission，37）和DEF（commission，38）。这有一个使用节点：USE（commission，41）。

表10-4　佣金的定义-使用路径

变　量	路径（开始，结束）节点	是否可行？	是定义清除吗？
commission	31，32	是	是
commission	31，33	是	否
commission	31，37	否	
commission	31，41	是	否
commission	32，32	是	是
commission	32，33	是	是
commission	32，37	否	—
commission	32，41	是	否
commission	33，32	否	
commission	33，33	是	是
commission	33，37	否	
commission	33，41	是	是
commission	36，32	否	
commission	36，33	否	
commission	36，37	是	是

（续）

变量	路径（开始，结束）节点	是否可行？	是定义清除吗？
commission	36，41	是	否
commission	37，32	否	
commission	37，33	否	
commission	37，37	是	是
commission	37，41	是	是
commission	38，32	否	
commission	38，33	否	
commission	38，37	否	
commission	38，41	是	是

10.1.7 定义–使用路径测试覆盖指标

上面介绍的程序分析的核心，是定义一组叫做Rapps-Weyuker数据流指标（Rapps，1985）。Rapps-Weyuker数据流指标的头三项等价于第9章介绍的三个E. F. Miller的指标：全路径、全边和全节点。其他数据流指标都假设所有程序变量都标识了定义节点和使用节点，并且关于各个变量都标识了定义-使用路径。在以下定义中，T是拥有变量集合V的程序P的程序图G（P）中的一个路径集合。计算变量的DEF节点集合与USE节点集合的叉积，对于定义–使用路径是不够的，前面已经讨论过，这种机械式的方法会产生不可行的路径。在以下定义中，我们假设定义/使用路径都是可行的。

定义

集合T满足程序P的全定义准则，当且仅当所有变量v ∈ V，T包含从v的每个定义节点到v的一个使用的定义清除路径。

定义

集合T满足程序P的全使用准则，当且仅当所有变量v ∈ V，T包含从v的每个定义节点到v的所有使用，以及到所有USE（v，n）后续节点的定义清除路径。

定义

集合T满足程序P的全谓词使用/部分计算使用准则，当且仅当所有变量v ∈ V，T包含从v的每个定义节点到v的所有谓词使用的定义清除路径，并且如果v的一个定义没有谓词使用，则定义清除路径导致至少一个计算使用。

定义

集合T满足程序P的全计算使用/部分谓词使用准则，当且仅当所有变量v ∈ V，T包含从v的每个定义节点到v的所有计算使用的定义清除路径，并且如果v的一个定义没有计算使用，则定义清除路径导致至少一个谓词使用。

定义

集合T满足程序P的全定义-使用路径准则，当且仅当所有变量v ∈ V，T包含从v的每个

定义节点到v的所有使用，以及到所有USE（v，n）后续节点的定义清除路径，并且这些路径要么有一次的环经过，要么没有环路。

这些测试覆盖指标有几个基于集合论的关系，在Rapps（1985）中被叫做“子假设”。这些关系在图10-3中给出。

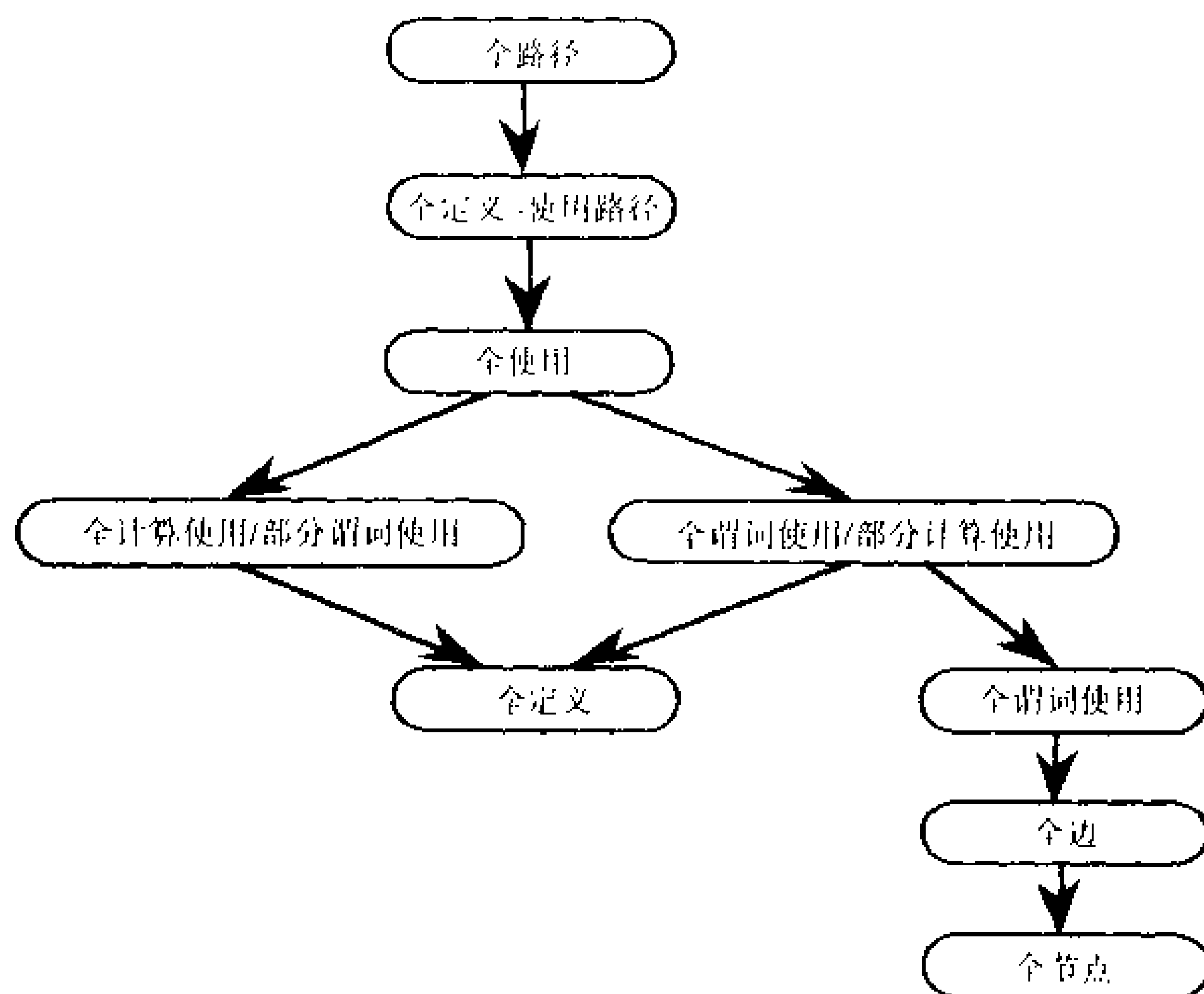


图10-3 数据流覆盖指标Rapps/Weyuker层次结构

现在我们有了一种介于（一般不能达到的）全路径指标和一般被认为是最低指标的全边之间的、更细化的结构性测试可能性。这些指标有什么意义呢？定义/使用测试提供一种检查缺陷可能发生点的严格和系统化方法。

10.2 基于程序片的测试

程序片在20世纪80年代初，在软件工程文献中曾经出现过一段时间。程序片最初是由Weiser（1985）提出的，Gallagher（1991）将其用做一种软件维护方法，后来又被Bieman（1994）用于量化功能的内聚性。有这种灵活性，部分原因是程序片概念具有自然、直观的清晰内涵。非正式地说，程序片是确定或影响某个变量在程序某个点上的取值的一组程序语句。这种概念还与其他学科一致。我们可以通过分片来研究历史：美国历史、欧洲历史、俄罗斯历史、远东历史、罗马历史等。这种历史片的交互方式与程序片交互方式之间具有很好的类比性。

我们首先给出程序片的定义。这里继续沿用定义-使用路径所使用的符号：程序P有程序图G（P）和一个程序变量集合V。Gallagher（1991）首先试图在定义中使P（G）中的节点引用语句片段。

定义

给定一个程序P和P中的一个变量集合V，变量集合V在语句n上的一个片，记做S（V，n），是P中对V中的变量值作出贡献的所有语句集合

列出S（V，n）中的元素会很麻烦，因为这些元素都是程序语句片段。列出P（G）中的语句片段编号会容易得多，因此我们对上述定义做一点变动（这不会引起集合论学者的不满）

定义

给定一个程序P和一个给出语句及语句片段编号的程序图G（P），以及P中的一个变量集合V，变量集合V在语句片段n上的一个片，记做S（V，n），是P中在n以前对V中的变量值作出贡献的所有语句片段编号的集合

片的思想，是将程序分成具有某种（功能）含义的组件。首先，需要解释一下定义的两个部分。这里的"在n以前"具有动态意义，因此片可获取对应该片内变量的程序执行时间行为。我们最终要开发出片的一种格（有向无环路图），格中的节点代表片，边代表子集关系。

"贡献"部分更复杂一些。在一定意义上，数据声明语句对变量值有影响。不过现在我们把所有非可执行语句排除在外。贡献概念可以部分地由Rapps（1985）所定义的谓词使用和计算使用澄清，但是我们还需要细化这些形式的变量使用。具体地说，USE关系适合五种形式的使用：

谓词使用　用在谓词（判断）中

计算使用　用在计算中。

输出使用　用于输出。

定位使用　用于定位（指针、下标）

迭代使用　迭代（内部计数器、循环指示）。

为了保持一致，我们标识两种定义节点：

输入定义　通过输入定义

赋值定义　通过赋值定义

现在先假设片S（V，n）是一个变量上的片，即集合V由单一变量v组成。如果语句片段n是v的一个定义节点，则n包含在该片中。如果语句片段n是v的使用节点，则n不包含在该片中。其他变量的谓词使用和计算使用（不是片集合V中的v），要包含其执行会影响变量v取值的扩展。作为一种指导方针，如果不管是否包含语句片段，v的值都保持不变，那么排除

该语句片段。

定位使用和迭代使用变量对于外部其他模块一般是不可见的，但是这类变量也常常出现错误。另一条在学术上有争议的建议：我们决定不把这些变量看做是“贡献”的一部分。这样，输出使用节点、定位使用节点和迭代使用节点，都不在片中。

10.2.1　举例

本书选用了佣金问题，因为佣金问题包含了有意思的数据流特性，而在三角形问题和NextDate问题中就没有这样的特性。研究下面的例子需要结合在定义-使用路径分析中使用过的佣金问题源代码。

变量locks上的片可说明为什么会很容易引入错误。这个片在节点14处有一个谓词使用，在节点16处有一个计算使用，并有两个定义，即节点13和19的输入定义。

S_1：S（locks，13）={13}
S_2：S（locks，14）={13，14，19，20}
S_3：S（locks，16）={13，14，16，19，20}
S_4：S（locks，19）={19}

变量stocks和barrels上的片很枯燥，都是完全局限在循环内部的很短的定义清除路径，因此不会受到循环迭代的影响。（把循环体看做是DD-路径。）

S_5：S（stocks，15）={13，14，15，19，20}
S_6：S（stocks，17）={13，14，15，17，19，20}
S_7：S（barrels，15）={13，14，15，19，20}
S_8：S（barrels，18）={13，14，15，18，19，20}

剩下的四个片可说明在片中如何出现重复。节点10是totalLocks的赋值定义，节点16既包含一个赋值定义也包含一个计算使用。S_{10}（13、14、24、19和20）中的剩余节点属于由locks控制的while循环。片S_{10}、S_{11}和S_{12}是相等的，因为节点21和27分别是totalLocks的一个输出节点和一个计算节点。

S_9：S（totalLocks，10）={10}
S_{10}：S（totalLocks，16）={10，13，14，16，19，20}
S_{11}：S（totalLocks，21）={10，13，14，16，19，20}

totalStocks和totalBarrels上的片很相似，在节点11和12处被赋值定义初始化，然后在节点17和18处被赋值定义重新定义。同样地，（13、14、19和20）中的剩余节点也属于由locks控制的while循环。

S_{12}：S（totalStocks，11）={11}

S_{13}：S（totalStocks，17）={11，13，14，15，17，19，20}

S_{14}：S（totalStocks，22）={11，13，14，15，17，19，20}

S_{15}：S（totalBarrels，12）={12}

S_{16}：S（totalBarrels，18）={12，13，14，15，18，19，20}

S_{17}：S（totalBarrels，23）={12，13，14，15，18，19，20}

下面六个片可说明由赋值语句（赋值定义）定义值的规则：

S_{18}：S（lockPrice，24）={7}

S_{19}：S（stockPrice，25）={8}

S_{20}：S（barrelPrice，26）={9}

S_{21}：S（lockSales，24）={7，10，13，14，16，19，20，24}

S_{22}：S（stockSales，25）={8，11，13，14，15，17，19，20，25}

S_{23}：S（barrelSales，26）={9，12，13，14，15，18，19，20，26}

sales和commission上的片很有意思，sales只有一个定义节点，即节点27处的赋值定义。sales上的其他片给出定义清除路径中的谓词使用、计算使用和输出使用。

S_{24}：S（sales，27）={7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27}

S_{25}：S（sales，28）={7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27}

S_{26}：S（sales，29）={7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27}

S_{27}：S（sales，33）={7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27}

S_{28}：S（sales，34）={7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27}

S_{29}：S（sales，37）={7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27}

S_{30}：S（sales，38）={7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27}

把片 S_{24} 看做是计算使用变量上的片"组件"组成的，可以写为 $S_{24}=S_{10}\cup S_{13}\cup S_{16}\cup S_{21}\cup S_{22}\cup S_{23}$。请注意这种形式与直觉的一致性：如果销售额的值是错的，则首先检查它是如何计算的，如果计算没有问题，则检查这些组件是如何计算的

最后一切都（字面上）汇总到commission的片上。六个赋值定义节点用于佣金（对应于前面找出的六个定义-使用路径）。佣金的三个计算由IF、ELSE IF逻辑中sales的谓词使用控制，产生计算commission的三个片"路径"。

S_{31}: S（commission，31）= {31}

S_{32}: S（commission，32）= {31，32}

S_{33}: S（commission，33）= {7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27，29，30，31，32，33}

S_{34}: S（commission，36）= {36}

S_{35}: S（commission，37）= {7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27，36，37}

S_{36}: S（commission，38）= {7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27，29，34，38}

不管进行什么计算，最后都会汇集到最后片上：

S_{37}: S（commission，41）= {7，8，9，10，11，12，13，14，15，16，17，18，19，20，24，25，26，27，29，30，31，32，33，34，35，36，37，38}

片信息可以提高我们的判断能力。图10-4所示的格是一种有向无环路图，其中的片是节点，边表示合适的子集关系。

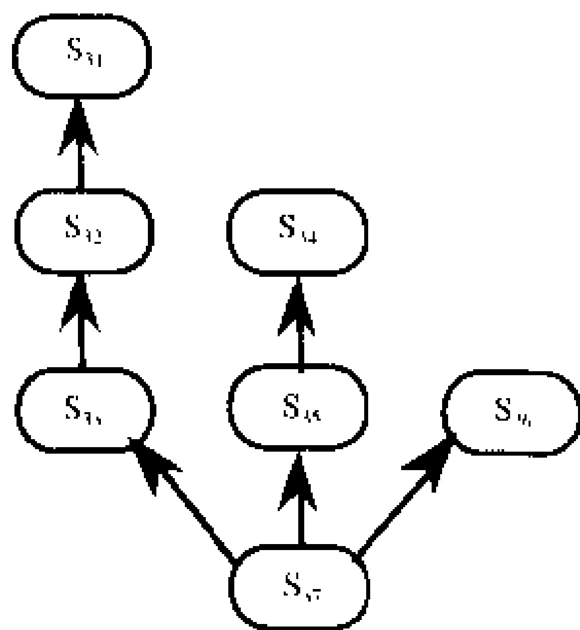


图10-4　commission上的片格

这种格使片节点的位置大致对应其源代码所在的位置。定义清除路径<43，51>、<48，51>和<50，51>对应的边表示片S_{36}、S_{38}和S_{39}是片S_{40}的子集。图10-5给出了整个程序的片格。为了清晰起见，删除了一些（相互一样的）片。

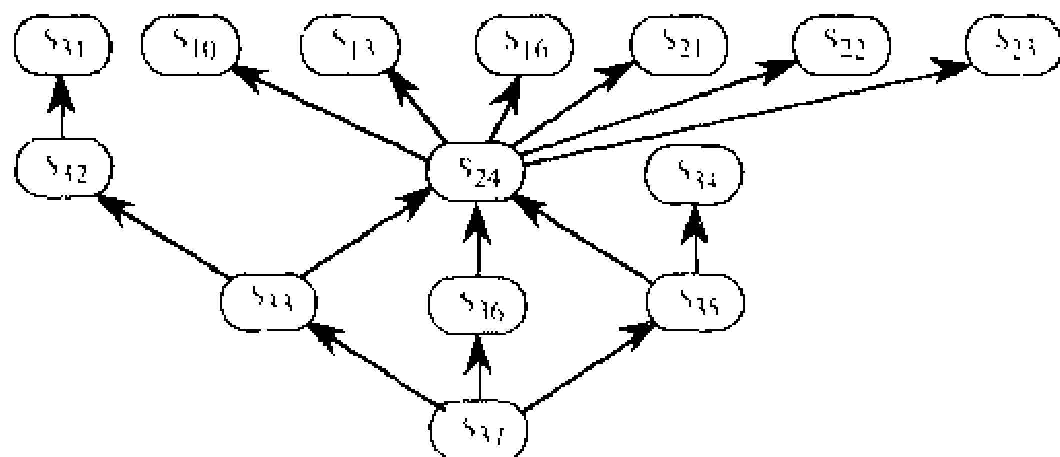


图10-5 sales和commission上的格

10.2.2 风格与技术

当我们通过感兴趣的片分析程序时，可以将注意力集中到感兴趣的部分，而不考虑无关的部分。采用定义-使用路径分析时不是这样做的，定义-使用路径是包含可能没有意思的语句和变量的序列。在讨论分析手段之前，首先讨论一下“好的风格”。我们本来可以把这些风格规则结合到定义中，但是会使定义更复杂。

1. 永远不要在不出现在语句片段n里的V的变量v上建立片S（V，n）。根据定义允许这样做，但是这是一种坏的实践。例如，假设在节点27处的locks变量上定义一个片。定义这种片对于跟踪程序中的所有点上的所有变量是必要的。

2. 在一个变量上建立片。片S（V，n）中的集合V可以包含多个变量，有时这样的片也是有用的。

V = {num_locks，num_stocks，num_barrels}

其中的片S（V，36）包含片S（{sales}，36），语句36除外。这两个片非常相似，因此为什么用计算使用定义片呢？

3. 对所有赋值定义节点都建立片。当变量被赋值语句计算时，该语句处的变量上的片应该包含在该计算中所使用的变量的所有定义-使用路径（部分）。片（{sales}，36）就是赋值定义片的一个很好的例子。

4. 对谓词使用节点建立片。当变量在谓词中使用时，该判断语句处变量上的片显示谓词变量如何得到值。这对于判断密集型程序非常有用，例如三角形程序和NextDate程序。

5. 非谓词使用节点上的片并不是很有意思。上面第2点讨论过计算使用片，我们看到这些片与赋值定义片有很大冗余。输出使用变量上的片永远可以表示为输出使用变量所有赋值定义（和输入定义）上的片。迭代使用和输出使用变量上的片在调试期间很有用，如果要用于所有测试，则测试工作量会急剧增加。

6. 考虑使片可编译。在片定义中并不要求语句集合是可编译的，但是如果使片可编译，则意味着编译器指令和数据说明语句集合是每个片的子集。

如果将相同的语句集合加到为佣金问题建立的所有片中，则原来的格不会受到影响，但是每个片都是可独立编译的（因此是可执行的） 在第1章中，我们指出好的测试实践或导致更好的程序设计实践 以下是一个很好的例子 按照可编译片考虑程序的开放 如果这样做了，就可以编写出片的代码并可立即测试 然后可以编写和测试其他片，再合并（有时叫做“片接合”）为相当坚实的程序 可尝试采用这种方法编写佣金程序

10.3 指导方针与观察

数据流测试显然适用于计算密集的程序 结果，在控制密集的程序中，如果要计算控制变量（谓词使用），则数据流测试也适用 定义/使用路径和片的定义，使我们能够非常准确地描述我们要测试的程序部分 学术界开发的原型工具可用来支持这些定义，但是还没有商业化 已经有了一些东西，可以找到能够在屏幕上显示片的程序设计语言编译器，大多数调试工具也使用户在单步执行程序时，能够“观察”到一定的变量 以下给出的内容实践已经证明很有用，尤其是要测试复杂的模块。

1. 片不能很好地映射到测试用例上（因为其他非相关代码仍然在执行路径中） 另一方面，片提供消除变量之间交互的一种方便手段。使用片合成方法来开发困难的代码部分，在把这些片接合（合成）到其他片上之前进行单独测试。

2. 片的相对补可提供诊断能力 集合B关于另一个集合A的相对补，是A中所有不在B中的元素集合，叫做A－B。考虑相对补集S（commission，48）－S（sales，35）：

S（commission，48）＝{3，4，5，36，18，19，20，23，24，25，26，27，34，38，39，40，44，45，47}

S（sales，35）＝{3，4，5，36，18，19，20，23，24，25，26，27}

S（commission，48）－S（sales，35）＝{34，38，39，40，44，45，47}

如果代码48行的commission存在问题，则可以将程序分成两部分：34行的sales计算以及35和48行的commission计算。如果34行的sales没有问题，则问题一定出在相对补中；如果不是，则可能是其他部分的问题。

3. 在片和DD-路径之间存在多对多关系：一个片中的语句可以在多条DD-路径中，一条DD-路径中的语句可能在多个片中 精心选择片的补可能与DD-路径相同 例如，考虑S（commission，40）－S（commission，37）。

4. 如果要开发片的格，假设第一条语句是片很方便，这样片格会永远在一个根节点上终止 用双向箭头表示相等的片。

5. 片可以反映出定义/引用信息 考虑totalLocks上的以下片：

S_9: S (totalLocks, 10) = {10}

S_{10}: S (totalLocks, 16) = {10, 13, 14, 16, 19, 20}

S_{11}: S (totalLocks, 21) = {10, 13, 14, 16, 19, 20}

如果片是相等的，则对应的路径就是定义清除的

10.4 参考文献

Clarke, Lori A. et al., A formal evaluation of data flow path selection criteria, *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 11, pp. 1318–1332, November 1989.

Rapps, S. and Weyuker, E.J., Selecting software test data using data flow information, *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pp. 367–375, April, 1985.

10.5 练习

1. 请考虑用DD-路径表示定义-使用路径的静态任意性和动态歧义性，首先考虑在sales的定义-使用路径p12、p13和p14中可找到什么DD-路径？

2. 尝试将基于DD-路径的测试覆盖指标合并到图10-2所示的Rapps/Weyuker层次结构中。

3. 请列出commission变量的定义-使用路径。

4. 请将片S_{17}表示为其他合适片的并。

5. 请找出以下程序片：

   a. S (commission, 28)

   b. S (sales, 23)

   c. S (commission, 37), S (commission, 29), S (commission, 28)

   d. S (totalLocks, 22)

   e. S (totalStocks, 22)

   f. S (totalBarrels, 22)

6. 请从代码行27~29，33、34、38、39中找出（关于销售额的）定义清除路径。

7. 请补充图10-5中的格。

8. 本章关于片的讨论，实际上是“反向片”，因为我们总是关心在程序的特定点上对变量值有贡献的程序部分。我们还可以考虑“正向片”，即使用该变量的程序部分。请对比正向片和定义-使用路径之间的异同。

第11章
结构性测试回顾

什么时候测试可以停止？以下是一些可能的答案：

1. 当时间用光时。
2. 当继续测试没有产生新失效时。
3. 当继续测试没有发现新缺陷时。
4. 当无法考虑新测试用例时。
5. 当回报很小时。
6. 当达到所要求的覆盖时。
7. 当所有缺陷都已经清除时。

但是，第一个答案太常见，第七个答案不能保证。这样测试工艺师就只能在中间的答案中选择。软件可靠性模型提供支持第二和第三种选择的答案，这些方法在业界都得到成功的使用。第四种选择不同一般，如果遵循前面讨论过的规则和指导方针，则这可能是一种好的答案。另一方面，如果是由于缺乏动力，则这种选择与第一种一样不幸。回报变小选择具有一定的吸引力：它指的是持续进行了认真的测试，并且所发现的新缺陷急剧降低，继续测试变得很昂贵，并且可能不会发现新的缺陷。如果能够确定剩余缺陷的成本（或风险），则折衷考虑比较清晰且容易做出。（这是一个很大的疑问。）剩下的是覆盖答案，而且是相当不错的答案。本章将介绍将结构性测试用做对功能性测试的交叉检查，如何能够产生强有力的结果。首先，我们（通过例子）说明功能性测试用例的漏洞和冗余问题，然后开发一些测试效率指标。由于这些指标都是通过结构覆盖表述的，因此对漏洞和冗余问题能够有明确的解答。然后问题就变成要使用哪些覆盖指标。在业界实践中最常用的是DD-路径。

11.1 漏洞与冗余

功能性测试的漏洞和冗余问题，在三角形问题中非常突出。这里我们使用（笨拙的）传统实现，主要因为这种实现在测试文献中最常使用（Brown，1975；Pressman，1982）。前面提到过，这个实现有11条可行路径，如表11-1所示。稍后将使用路径名称，节点编号如图11-1所示。

表11-1 三角形程序中的路径

路径	节点序列	描述
p1	1 – 2 – 3 – 4 – 5 – 6 – 7 – 13 – 16 – 18 – 20	等边三角形
p2	1 – 3 – 5 – 6 – 7 – 13 – 16 – 18 – 19 – 15	等腰三角形（b = c）
p3	1 – 3 – 5 – 6 – 7 – 13 – 16 – 18 – 19 – 12	非三角形（b = c）
p4	1 – 3 – 4 – 5 – 7 – 13 – 16 – 17 – 15	等腰三角形（a = c）
p5	1 – 3 – 4 – 5 – 7 – 13 – 16 – 17 – 12	非三角形（a = c）
p6	1 – 2 – 3 – 5 – 7 – 13 – 14 – 15	等腰三角形（a = b）
p7	1 – 2 – 3 – 5 – 7 – 13 – 14 – 12	非三角形（a = b）
p8	1 – 3 – 5 – 7 – 8 – 12	非三角形（a + b ≤ c）
p9	1 – 3 – 5 – 7 – 8 – 9 – 12	非三角形（b + c ≤ a）
p10	1 – 3 – 5 – 7 – 8 – 9 – 10 – 12	非三角形（a + c ≤ b）
p11	1 – 3 – 5 – 7 – 8 – 9 – 10 – 11	不等边三角形

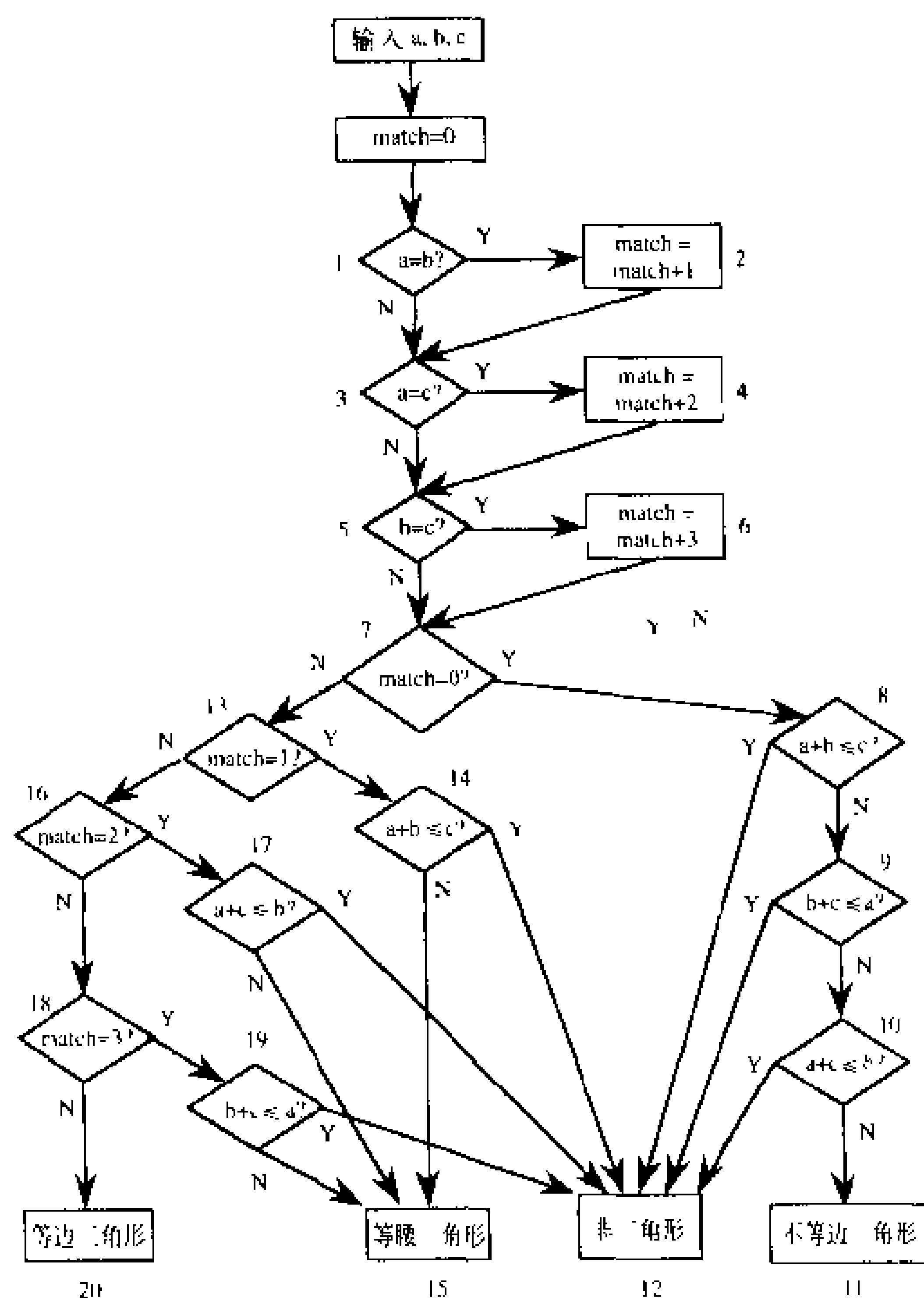


图11-1 传统三角形程序图

现在，假设使用边界值测试来定义测试用例。我们将给出一般和最坏情况的公式化。表11-2使用功能性测试的一般边界值形式所生成的测试用例，最后一列给出测试用例将经过的（图11-1中的）路径。

表11-2　一般值的路径覆盖

测试用例	a	b	c	预期输出	路径
1	100	100	1	等腰三角形	p6
2	100	100	2	等腰三角形	p6
3	100	100	100	等边三角形	p1
4	100	100	199	等腰三角形	p6
5	100	100	200	非三角形	p7
6	100	1	100	等腰三角形	p4
7	100	2	100	等腰三角形	p4
8	100	100	100	等边三角形	p1
9	100	199	100	等腰三角形	p4
10	100	200	100	非三角形	p5
11	1	100	100	等腰三角形	p2
12	2	100	100	等腰三角形	p2
13	100	100	100	等边三角形	p1
14	199	100	100	等腰三角形	p2
15	200	100	100	非三角形	p3

以下路径被覆盖：p1、p2、p3、p4、p5、p6、p7。路径p8、p9、p10、p11没有被覆盖。现在假设使用更有力的功能性测试技术，即最坏情况边界值测试。第5章曾经讨论过，它将产生125个测试用例，这里归纳在表11-3中，使读者能够看到冗余路径覆盖范围。合在一起，125个测试用例提供全路径覆盖，但是冗余很严重。

表11-3　最坏情况值路径覆盖

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10	p11
一般值	3	3	1	3	1	3	1	0	0	0	0
最坏情况	5	12	6	11	6	12	7	17	18	19	12

11.2　用于方法评估的指标

确信了功能性测试方法确实存在漏洞和冗余问题之后，我们可以开发一些将功能性测试技术有效性和结构性测试进展指标关联的指标。功能性测试技术永远会产生一组测试用例，结构性测试指标永远以可计算的内容表示，例如程序路径数，决策到决策路径（DD-路径）数，或片数。

在以下定义中，我们假设功能性测试技术M生成m个测试用例，并且根据标识被测单元

中的s个元素的结构性测试指标S来跟踪这些测试用例。当执行m个测试用例时，会经过n个结构性测试元素。

定义

方法M关于指标S的覆盖是n与s的比值，记做C（M，S）。

定义

方法M关于指标S的冗余是m与s的比值，记做R（M，S）

定义

方法M关于指标S的净冗余是m与n的比值，记做NR（M，S）。

下面解释这些指标：覆盖指标C（M，S）处理漏洞问题。如果这个值低于1，则说明该指标在覆盖上存在漏洞。请注意，如果C（M，S）=1，则一定有R（M，S）=NR（M，S）冗余性指标很明显，取值越大，冗余性越高。净冗余更有用，它指实际经过的元素，而不是要经过的总元素空间。将三种指标集合在一起，给出一种评估功能性测试有效性方法（特殊值测试除外）关于结构性测试指标的定量方法。不过这才只是工作的一半。我们实际需要的是要知道测试用例关于缺陷种类的有效性。但是，不能得到这类信息。通过选择关于我们预期（或最担心）的缺陷种类的结构性测试指标，可以接近这个目标。具体建议请参阅第9章和第10章末的指导方针部分。

一般来说，结构性测试指标越精细，会产生更多的元素（s越大），因此给定功能性测试方法通过更严格的结构性测试指标评估时有效性变得较低。这与直观感觉是一致的，并且可以通过例子证明。这些指标的最好可能取值都是1。表11-4归纳了将这些定义用于表11-3中的数据所得到的结果。结构性测试指标是程序路径，共有11条程序路径。（考虑不可行的路径是没有意义的，因为测试用例永远不会经过不可行路径。）表11-5给出的是佣金问题的类似结果。

表11-4 用于三角形程序的指标

方法	m	n	s	C（M，S）=n/s	R（M，S）=m/s	NR（M，S）=m/n
一般值	15	7	11	0.64	1.36	2.14
最坏情况	125	11	11	1.00	11.36	11.36
目标	s	s	s	1.00	1.00	1.00

表11-5 用于佣金问题的指标

方法	m	n	s	C（M，S）=n/s	R（M，S）=m/s
输出边界值	25	11	11	1	2.27
决策表	3	11	11	1	0.27
DD-路径	25	11	11	1	2.27
定义-使用路径	25	33	33	1	0.76
片	25	40	40	1	0.63

图11-2和11-3分别表示的是测试覆盖项（定义中的s）数量的趋势线，以及将覆盖项标识为结构性测试方法的函数的作用。我们不再满足于功能性测试方法的折衷考虑，因为这两张图说明选择合适结构性测试覆盖指标的重要性。如果重新研究第8章的例子就会理解这一点。

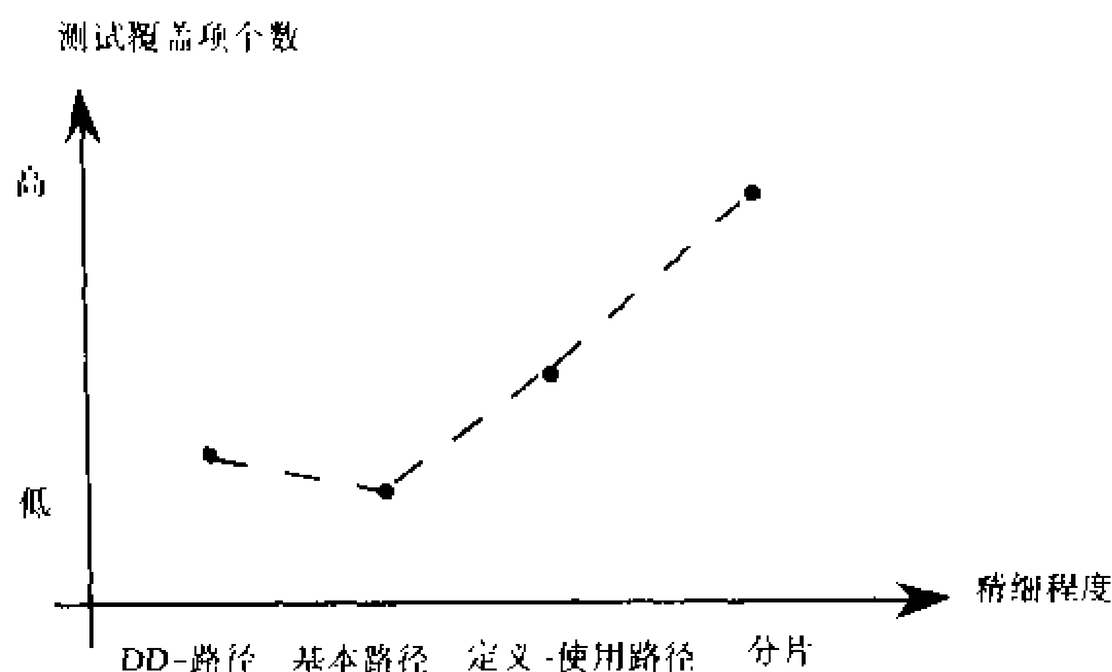


图11-2 测试覆盖项的趋势

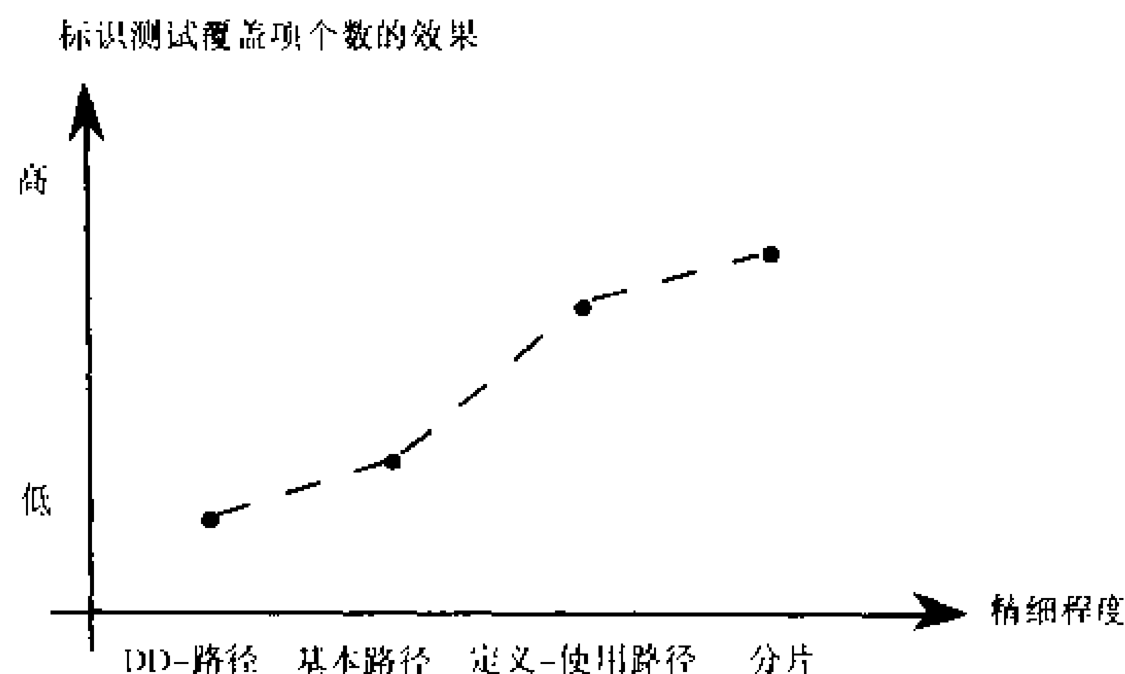


图11-3 测试方法作用的趋势

11.3 重温案例研究

下面继续我们的功能性测试回顾（第8章）的假想保险金程序案例研究。从这个伪代码实现完成的错误检查非常少这一点看，它是最低的。这个实现的程序图如图11-4所示。

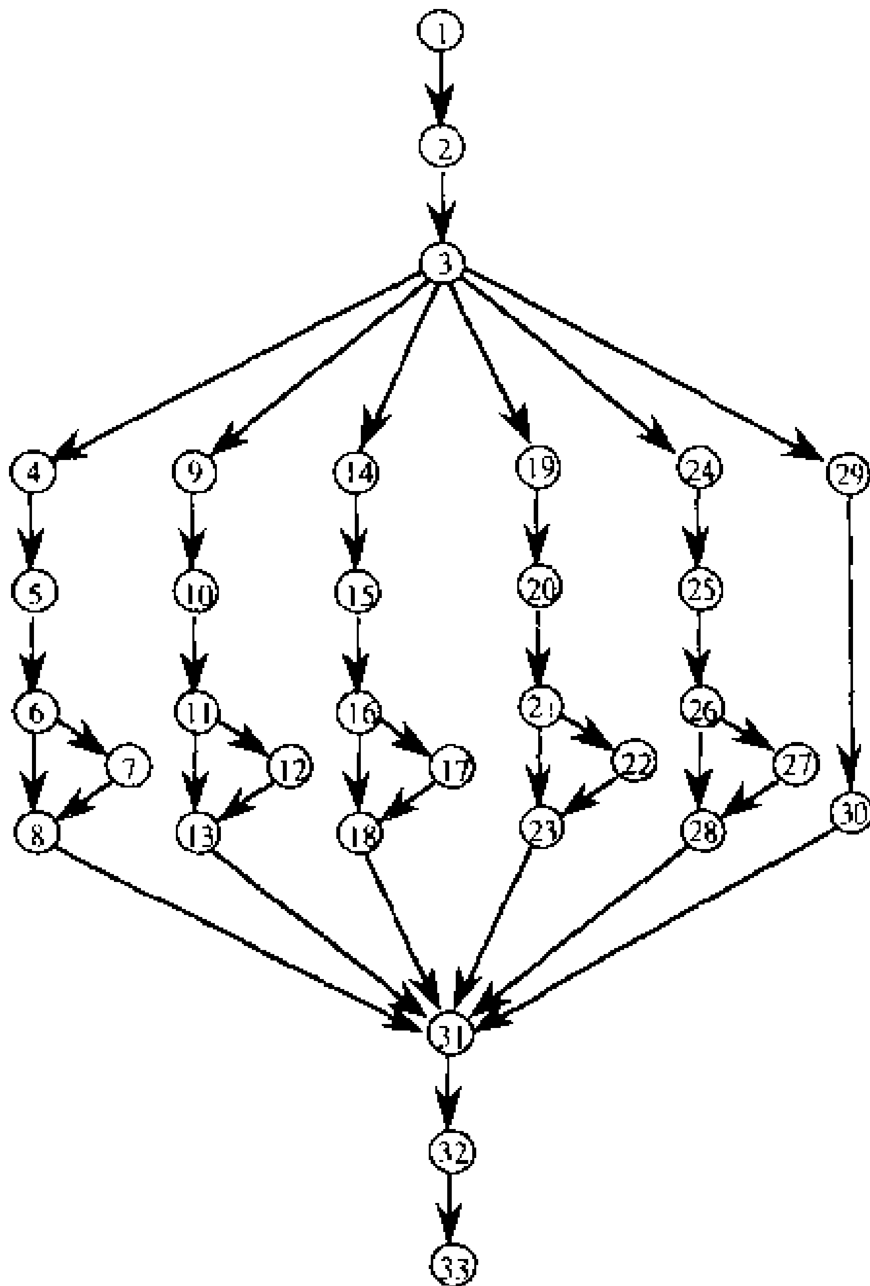


图11-4 保险金程序的程序图

Pseudo-code for the Insurance Premium Program

```
Dim driverAge, points As Integer
Dim baseRate, premium As Real

Input(baseRate, driverAge, points)
premium = 0
Select Case driverAge
Case 1: 16 ≤ driverAge < 20
    ageMultiplier = 2.8
    If points < 1 Then
        safeDrivingReduction = 50
    End If
Case 2: 20 ≤ driverAge < 25
    ageMultiplier = 1.8
    If points < 3 Then
        safeDrivingReduction = 50
    End If
Case 3: 25 ≤ driverAge < 45
    ageMultiplier = 1#
    If points < 5 Then
        safeDrivingReduction = 100
```

```
18.     End If
19. Case 4: 45 ≤ driverAge < 60
20.     ageMultiplier = 0.8
21.     If points < 7 Then
22.         safeDrivingReduction = 150
23.     End If
24. Case 5: 60 ≤ driverAge < 120
25.     ageMultiplier = 1.5
26.     If points < 5 Then
27.         safeDrivingReduction = 200
28.     End If
29. Case 6: Else
30.         Output("Driver age out of range")
31. End Select
32. premium = baseRate * ageMultiplier - safeDrivingReduction
33. Output(premium)
```

保险金程序程序图的复杂度是V（G）= 11，有11条可行程序执行路径，如表11-6所示。

表11-6　保险金程序中的路径

路　径	节点序列
p1	1 – 2 – 3 – 4 – 5 – 6 – 8 – 31 – 32 – 33
p2	1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 31 – 32 – 33
p3	1 – 2 – 3 – 9 – 10 – 11 – 13 – 31 – 32 – 33
p4	1 – 2 – 3 – 9 – 10 – 11 – 12 – 13 – 31 – 32 – 33
p5	1 – 2 – 3 – 14 – 15 – 16 – 18 – 31 – 32 – 33
p6	1 – 2 – 3 – 14 – 15 – 16 – 17 – 18 – 31 – 32 – 33
p7	1 – 2 – 3 – 19 – 20 – 21 – 23 – 31 – 32 – 33
p8	1 – 2 – 3 – 19 – 20 – 21 – 22 – 23 – 31 – 32 – 33
p9	1 – 2 – 3 – 24 – 25 – 26 – 28 – 31 – 32 – 33
p10	1 – 2 – 3 – 24 – 25 – 26 – 27 – 28 – 31 – 32 – 33
p11	1 – 2 – 3 – 29 – 30 – 31 – 32 – 33

如果花一些时间研究第8章各种功能性测试用例集合的伪代码，就会得到如表11-7所示的结果。

表11-7　保险金程序中的功能性测试方法路径覆盖

图　号	方　法	测试用例	所覆盖的路径
8-7	边界值	25	p1、p2、p7、p8、p9、p10
8-8	最坏情况边界值	273	p1、p2、p3、p4、p5、p6、p7、p8、p9、p10
8-9	弱等价类	5	p2、p4、p6、p8、p9
8-9	强等价类	25	p1、p2、p3、p4、p5、p6、p7、p8、p9、p10
8-10	决策表	10	p1、p2、p3、p4、p5、p6、p7、p8、p9、p10
8-11	混合	25	p1、p2、p3、p4、p5、p6、p7、p8、p9、p10、p11

通过结构性测试研究，现在对这些问题会有更深的认识。其中的漏洞和冗余问题很明显，只有通过混合方法得到的测试用例才能够产生完全的路径覆盖。将这25个测试用例结果与产

生相同数量测试用例的其他两种方法做比较，会有所启发：25个边界值测试用例只覆盖六条可行执行路径，而25个强等价类测试用例覆盖了10条可行执行路径。下一种差别是case语句中的条件覆盖。每个判断都是形式为a < = x < b的复合条件。能够产生可执行这些极端值的测试用例的方法，只有最坏情况边界值（273个）测试用例和混合方法（25个）测试用例差别真大！

11.3.1 基于路径的测试

由于程序图是无环路的，因此只存在有限条路径，对于这个例子是11。最佳选择是保留执行每条路径的测试用例，自然会达到语句和DD-路径覆盖要求。复合条件谓词意味着多条件覆盖，这只能通过最坏情况边界值测试用例和混合测试用例实现，不能使用其他基于路径的覆盖指标。

11.3.2 数据流测试

这个问题的数据流测试很枯燥，driverAge、points和safeDrivingReduction变量都出现在六个定义清除的定义-使用路径中。driverAge和points的“使用”都是谓词使用。第10章曾经提到过，全路径准则意味着全低层数据流覆盖。

11.3.3 片测试

片测试也没有提供多少启发。有四个有意思的片（没有列出EndIf）：

S（safeDrivingReduction，32）={1，3，4，6，7，9，11，12，14，16，17，19，21，22，24，26，27，31}

S（ageMultiplier，32）={1，3，4，5，9，10，14，15，19，20，24，25，31}

S（baseRate，32）={1}

S（Premium，31）={2}

这些片的并（加上EndIf语句）是整个程序，通过基于片的测试得到的仅有启发，如果实现发生在第32行，safeDrivingReduction和ageMultiplier上的片将程序分解为两个不相交的片，这会简化缺陷隔离工作。

11.4 参考文献

Pressman, Roger S., *Software Engineering. A Practitioner's Approach*. McGraw-Hill, New York, 1982.

Brown, J.R. and Lipov, M., Testing for Software Reliability, *Proceedings of the International Symposium on Reliable Software,* Los Angeles, pp. 518–527, April 1975.

11.5 练习

1. 请使用第2章介绍的结构化实现和第9章介绍的DD-路径，重新分析三角形问题的漏洞和冗余问题。

2. 请对练习1的研究计算覆盖、冗余和净冗余指标。

3. 保险金程序伪代码没有检查低于16岁和（不大可能）超过120岁的驾驶员。Else子句（用例6）会捕获这些问题，但是输出消息不很具体。此外，输出语句不受驾驶员年龄检查的影响。哪些功能性测试手段可以发现这类缺陷？哪种结构性测试覆盖会发现这类缺陷？

第四部分

集成与系统测试

第12章
测试层次

本章构建第四部分的背景环境，以便研究传统软件的集成和系统测试。我们的直接目标是确定这些层次测试的含义。第1章曾标识了与软件开发瀑布式模型对称的一个有三个层次（单元、集成和系统）的简化视图。几十年来，这种测试观点相对成功，但是其他生命周期模型的出现要求重新研究这些测试观点。我们首先介绍传统瀑布模型，主要是因为它已经被很多人接受，并具有类似的表达能力。为了使讨论比较具体，还是使用自动柜员机的例子。

在第四和第五部分中，我们还要对思路做重要调整，将更注重如何表示被测试项，因为表达能力会限制我们标识测试用例的能力。研究一下有关软件测试的（专门或学术）主要会议上发表的论文，就会发现有关规格说明模型的论文与技术，与有关测试技术的论文几乎一样多。

12.1 测试层次的传统观点

传统软件开发模型是V型瀑布模型，如图12-1所示，强调测试的基本层次。在这种观点中，一个开发阶段产生的信息，构成该层次测试用例标识的基础。这里没有矛盾：我们当然也希望系统测试用例以某种方式与需求规格说明相关、单元测试用例从单元详细设计中导出。有两点需要指出：这里显然假设的是功能性测试，指自底向上的测试顺序。这里的"自底向上"是指抽象层次，第13章的自底向上也指单元集成（和测试）的顺序。

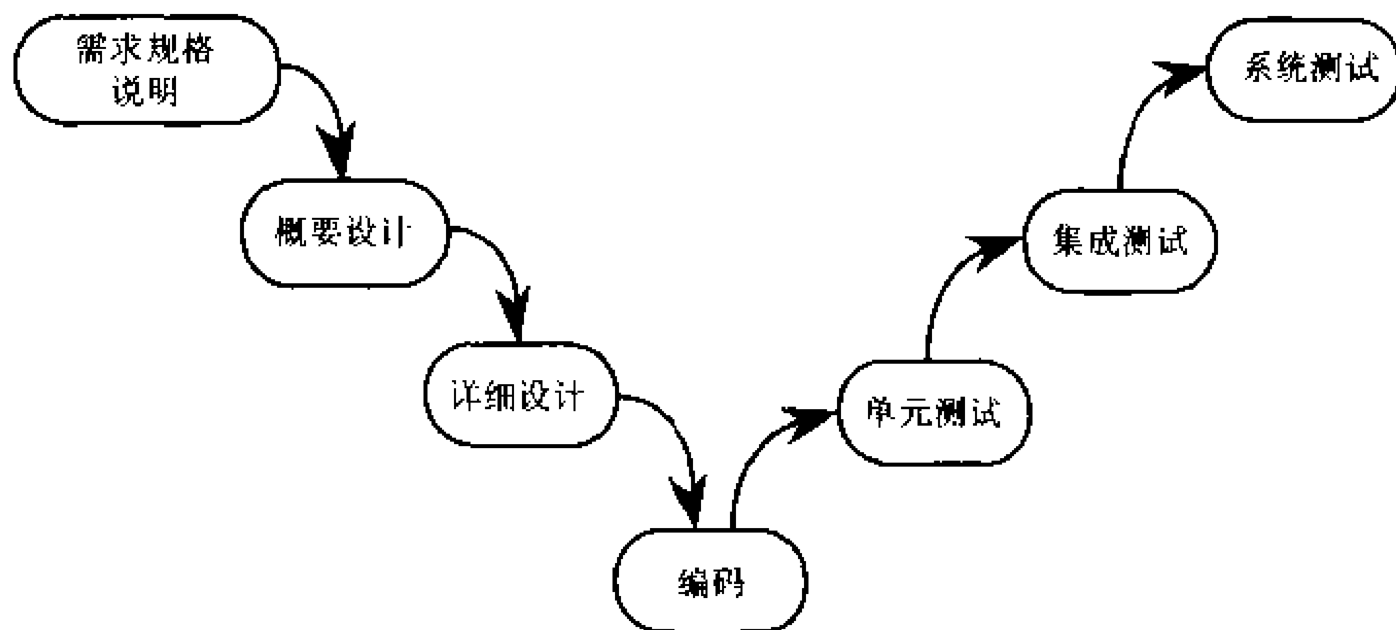


图12-1 瀑布式生命周期

在传统的三个测试层次（单元、集成和系统）中，单元测试是被理解得最好的。本书第

二、第三部分所讨论的测试理论和技术都可直接用于单元测试。对系统测试的理解要好于集成测试，但是两种测试都需要澄清。自底向上方法有这样的观点：测试单个组件，然后将这些组件集成到子系统中，直到测试完整个系统。系统测试应该是客户（或用户）能够理解的活动，常常与客户验收测试结合在一起进行。一般来说，系统测试是功能性测试，而不是结构性测试，主要是因为没有高层结构表示法。

瀑布模型与通过功能分解进行的自顶向下开发和设计密切相关。概要设计的最终结果，是将整个系统的功能分解为功能组件的树型结构。图12-2给出了自动柜员机（ATM）系统的部分功能分解。通过这种分解，自顶向下的集成要从主程序开始，检查对三个下层过程（“终端I/O”、“管理会话”和“引导事务处理”）的调用。按照树型结构向下，要测试“管理会话”过程，然后是“卡输入”、“PIN输入”和“选择事务处理”过程。对于每个测试用例，下层单元的实际代码都由桩替代，桩是占据实际代码位置的一次性代码。自底向上的集成按相反顺序进行，首先从“卡输入”、“PIN输入”和“选择事务处理”过程开始，然后是上升到主程序。在自底向上集成中，高层单元被模拟过程调用的驱动器（另一种一次性代码）取代。“大爆炸”方法一次将所有单元放在一起，不使用桩和驱动器。不管采用哪种方法，传统集成测试的目标都是根据功能分解树集成以前测试过的单元。虽然这里把集成测试描述为一种过程，但是这种讨论提供不了多少测试方法或技术的信息。在讨论这些（实际）问题之前，需要讨论其他生命周期模型是否对集成测试有影响。

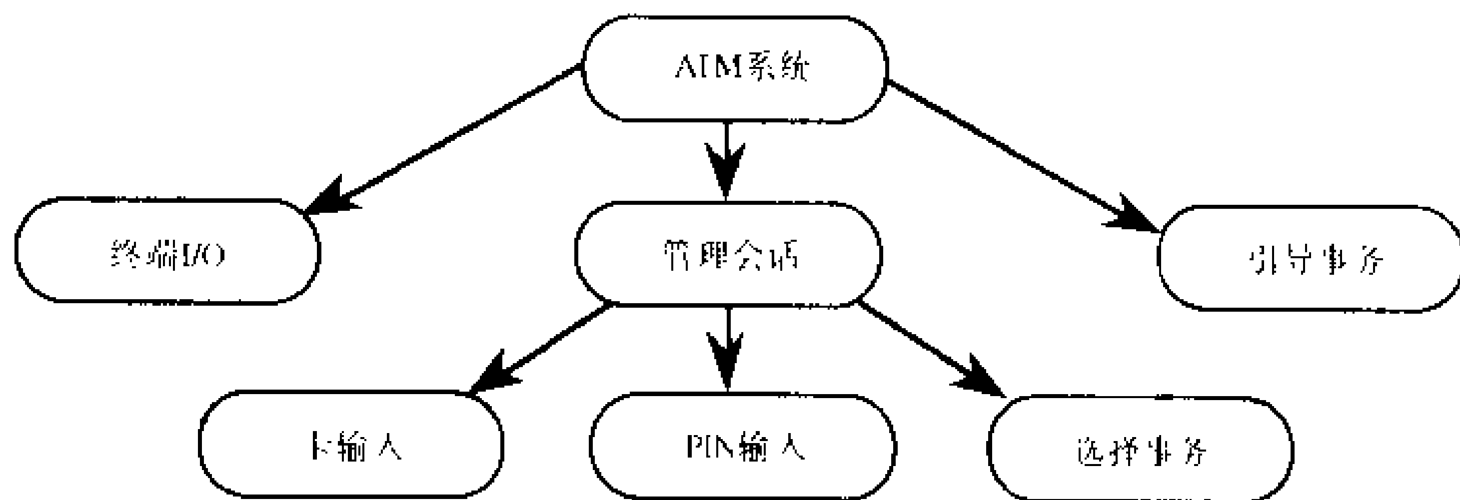


图12-2 ATM系统的部分功能分解

12.2 其他生命周期模型

20世纪80年代初以来，实践者提出了其他生命周期模型，以弥补软件开发传统瀑布模型存在的不足（Agresti，1986）。这些新模型的共同点是从功能分解转向强调合成。分解既能够很好地适合瀑布模型的自顶向下推进，也能够很好地适合自底向上的测试顺序。Agresti（1986）指出的瀑布式开发的主要弱点，是过于依赖这种整体范例。功能分解只有在系统被彻底理解之后才能很好地进行，过于强调分析几乎排除了综合，结果造成需求规格说明与

所完成系统的很长时间的分离。并且在此期间，没有机会得到客户的反馈。另一方面，合成更接近人们的工作方式：从已知和理解的东西开始，然后逐渐增加，可能还会删除不需要的部分。这可以与正雕塑和负雕塑做很好的类比。对于负雕塑，要逐步去掉不需要的材料，从数学家角度，可以这样看待米开朗基罗的大卫：先从一块大理石开始，凿掉非大卫的部分。正雕塑常常被蜡像馆采用。首先做出近似的核心形状，然后增加或去除蜡，直到得到所要的形状。考虑错误会产生的结果：对于负雕塑，整个工作必须推倒重来。（意大利佛罗伦萨博物馆保存了五六件这类失败的大卫雕像。）对于正雕塑，有问题的部分可以直接去除并替代。新模型以合成为核心，会对集成测试产生重要影响。

12.2.1 瀑布模型的新模型

瀑布模型的三种主流派生模型是：增量开发、进化开发和螺旋模型（Boehm，1988）。这些模型都涉及图12-3所示的一系列增加和构建。在一个构建中，会出现从详细设计到测试的一般瀑布阶段，但是有一点重要的不同：系统测试被分成两个步骤，即回归测试和累进测试。

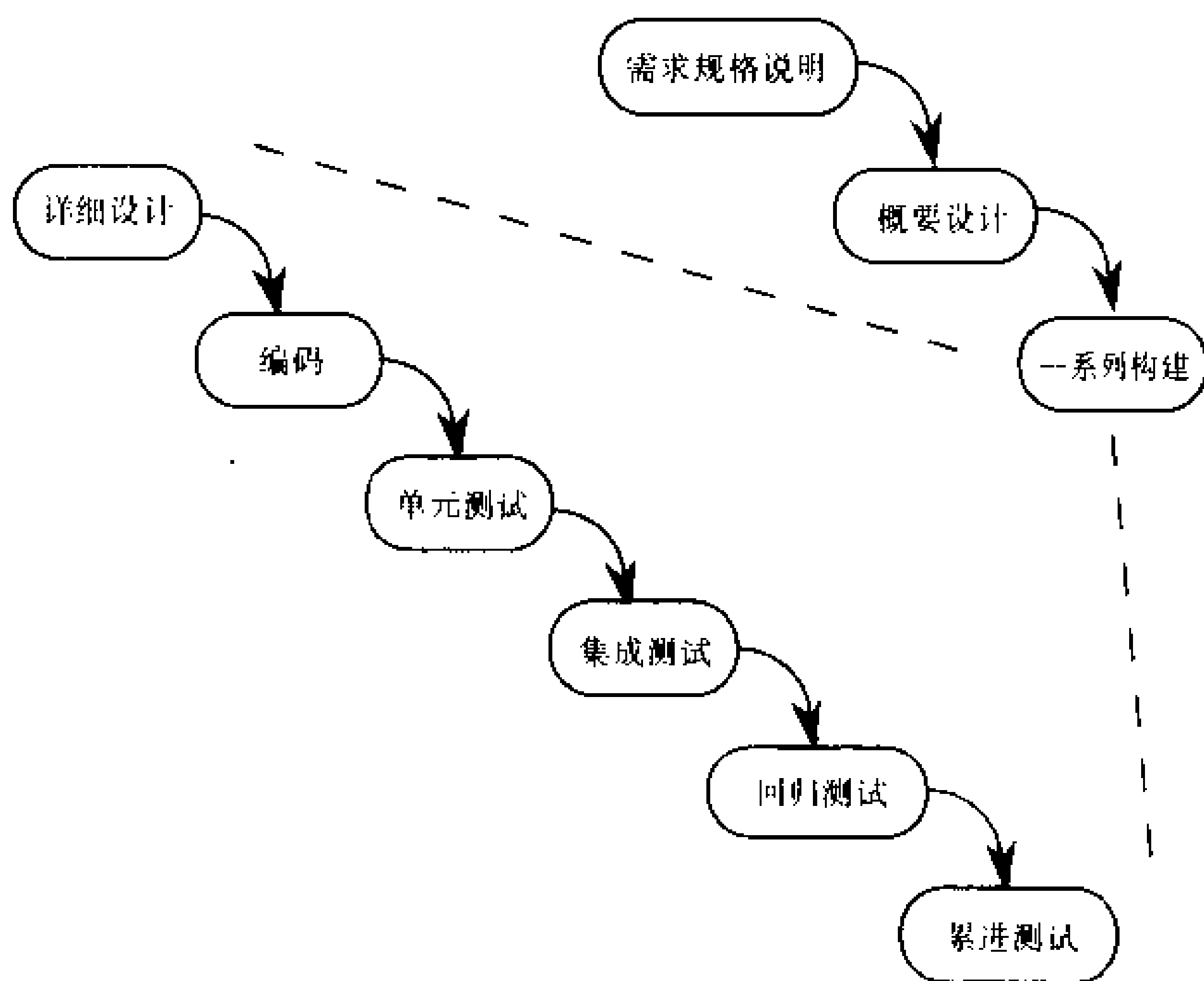


图12-3 有构建序列的生命周期

重要的是使概要设计成为一个集成阶段，而不是将这种高层设计分摊到一系列构建中（这样做常常会导致早期所做出的设计选择在后期构建中不合适。）由于概要设计仍然是单独的步骤，因此集成测试在新模型中不会受到影响——一系列构建的主要影响需要进行回归测试。回归测试的目标是保证在前一个构建中工作正常的功能，新增加了代码之后仍然工

作正常 累进测试假设回归测试是成功的，并且新功能是可以测试的.（我们倾向于认为新增加的代码代表进展，不代表回归.）回归测试对于一系列构建绝对必要，这是因为变更对现有系统具有众所周知的波及效应。（根据业界的统计，每五个变更就会有一个变更引入新缺陷.）

这三种新模型之间的差别是如何标识构建. 在增量开发模型中，划分构建的动机通常是均衡人员结构. 对于纯瀑布开发，从详细设计到单元测试，需要大量人力，大多数机构都不能支持这种快速的人员需求浮动，因此将系统划分为现有人力能够支持的构建 在进化开发中，仍然要假设一系列构建，但是只定义第一个构建 根据第一个构建标识后续构建，通常要与客户/用户设定的优先级一致，使得系统发展，以满足用户变化着的需求. 螺旋模型是快速原型和进化开发的结合，首先采用快速原型法定义一个构建，然后根据与技术有关的风险要素，做出后续决策。通过以上讨论可以看出，使概要设计成为一个集成步骤对进化模型和螺旋模型很困难. 由于概要设计不能成为一种集成活动，因此会对集成测试带来消极影响. 系统测试不受影响。

由于构建是一组可交付最终用户功能，因此这三种新模型的一个共同优点，是都产生比较早期的综合。这还会导致较早的客户反馈，因此能够缓解瀑布式开发的两个不足

12.2.2 基于规格说明的生命周期模型

还有两个模型变种可用来解决"完全理解"问题。（前面提到过，只有当系统被完全理解后功能分解才能成功.）如果系统不能被（客户或开发人员）充分理解，则功能分解一般是很危险的。快速原型法生命周期（请参见图12-4）通过急剧缩短规格说明到客户的反馈周期，以产生非常早的综合来解决这个问题. 不是构建最终系统，而是构建一种快速、不完善的原型，并用来启发客户提出反馈意见. 取决于客户的反馈意见，可能要有多个原型法周期. 一旦开发人员和客户就原型所表示的所需系统达成一致，开发人员就开始着手按照正确的规格说明构建系统。这时还可以使用任何瀑布模型的新模型

快速原型法对集成测试没有其他特别的要求，但是对系统测试有特别要求 需求在哪里？最后的原型是规格说明吗？系统测试用例如何反向跟踪到原型？对这类问题的一种好的回答是把原型法周期用做信息收集活动，然后以更传统的方式产生需求规格说明. 另一种可能是获取客户怎么使用原型，将他们的使用情况定义为对客户很重要的场景，然后使用这些场景作为系统测试用例. 快速原型法的重要贡献是为需求规格说明阶段引入操作（或行为）的观点。通常，需求规格说明技术强调的是系统的结构，而不是系统的行为 这很不幸，因为大多数客户都不大关心结构，而的确关心行为.

可执行规格说明（如图12-5所示）是快速原型法的一种扩展，通过这种方法，需求以某种可执行格式（例如有限状态机、状态图或Petri网）描述. 然后客户执行这种规格说明，以观察想实现的系统行为，并像快速原型法模型一样地提供反馈。

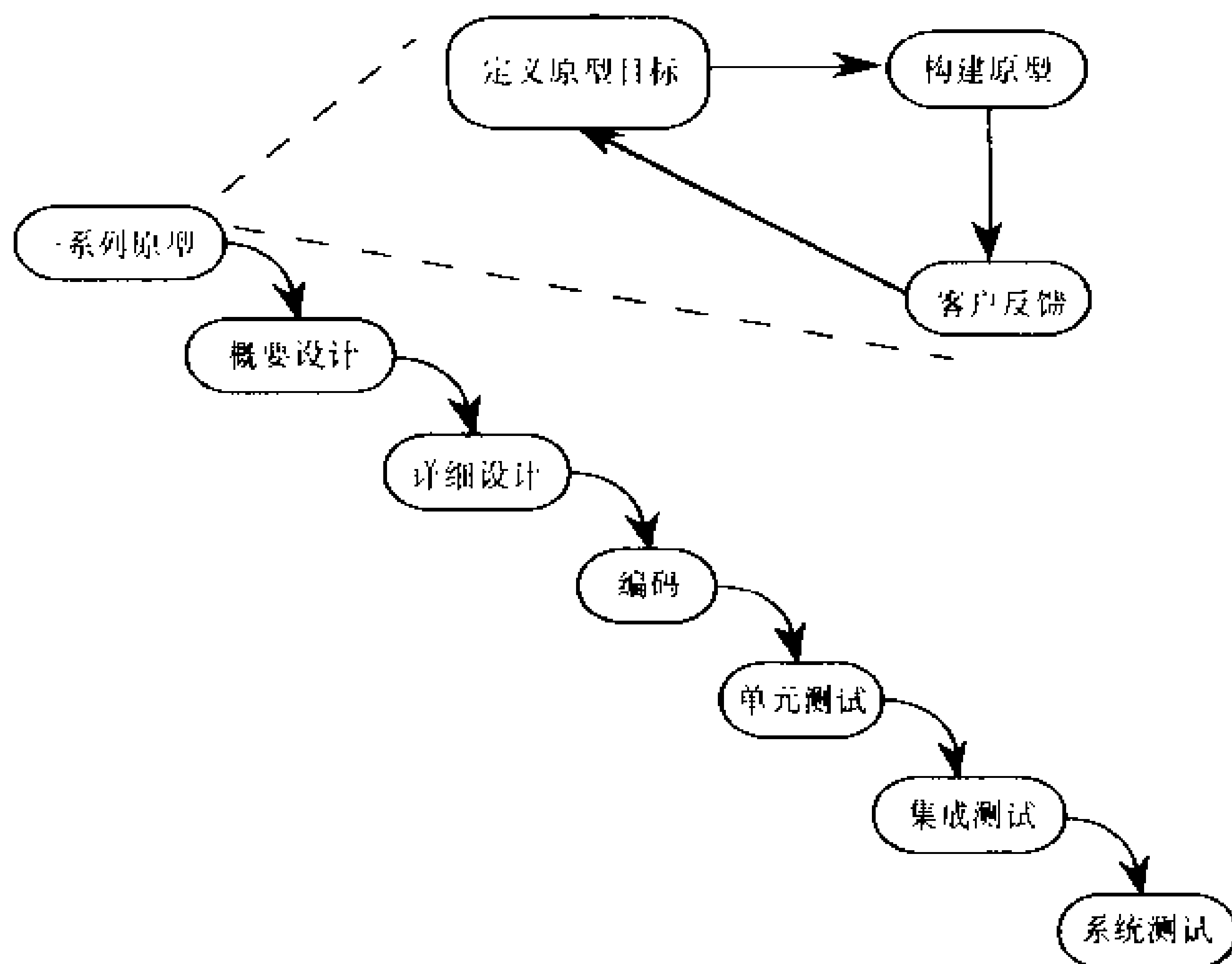


图12-4 快速原型法生命周期

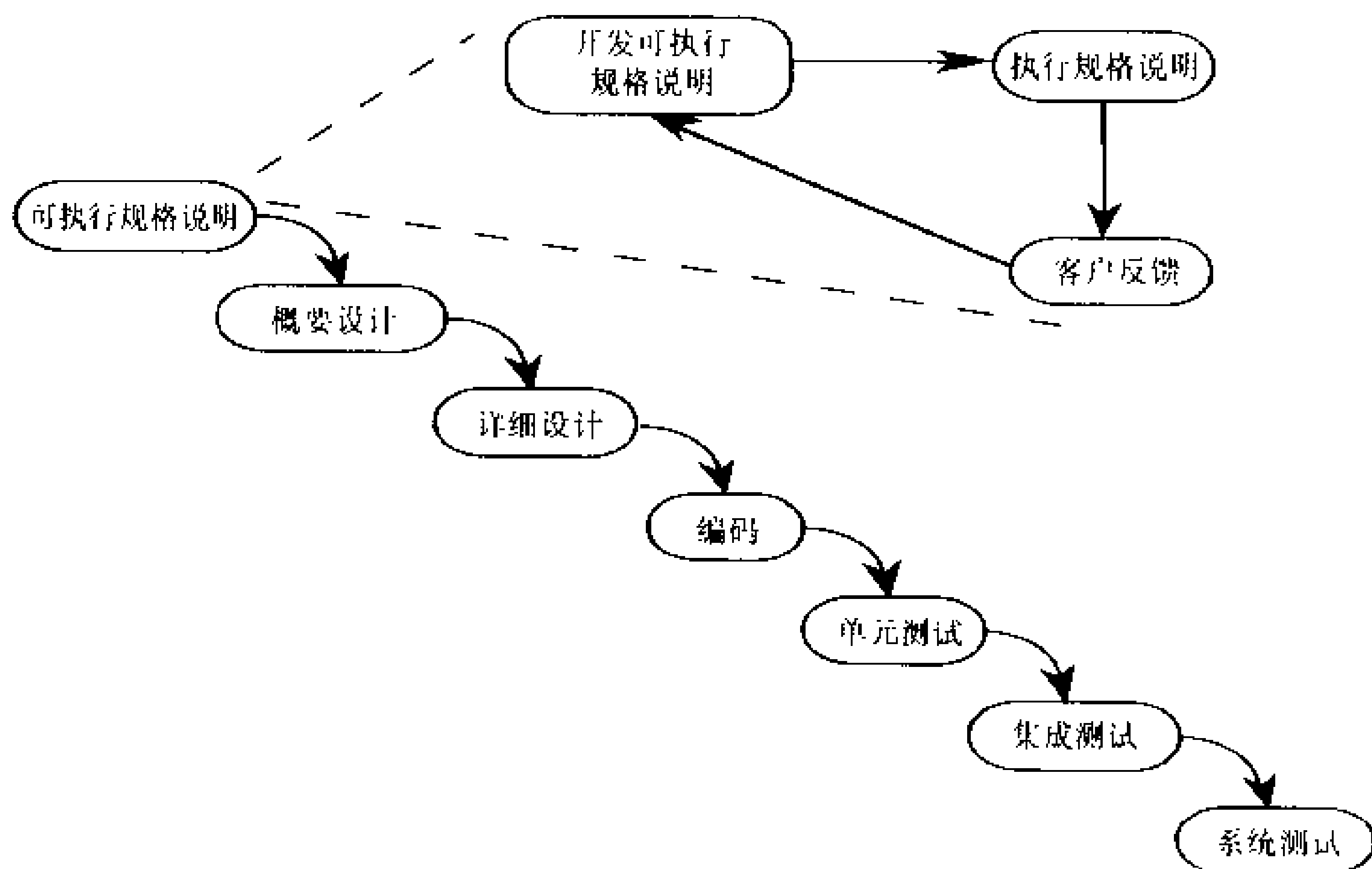


图12-5 可执行规格说明

同样，这种方法对集成测试也没有特别要求。有一点很大的差别是，需求规格说明文档是明确的，这与原型不同。更重要的是，从可执行规格说明中导出系统测试用例常常是一

种机械过程，第15章将讨论这一点。尽管开发可执行规格说明需要更多的工作，但是通过减少生成系统测试用例的工作量可以得到部分补偿。还有一点重要差别：如果系统测试根据可执行规格说明进行，则在系统层上可得到一种结构性测试的有意义的形式

12.3 ASTM系统

第四部分将把讨论与一个高层例子，即简单自动柜员机（SATM）系统关联起来。这里开发的版本是对Topper（1993）的一种改进，要围绕如图12-6所示的15个屏幕构建系统。这是一种经过很大精简的系统，商业化ATM系统有数百个屏幕和大量暂停。SATM终端的草图如图12-7所示，除了显示屏幕之外，这种终端还包括功能键B1、B2和B3，带有取消键的数字小键盘，用于进出打印收据和ATM卡的槽，以及用于存入和提取现金的通道



图12-6 SATM系统的屏幕

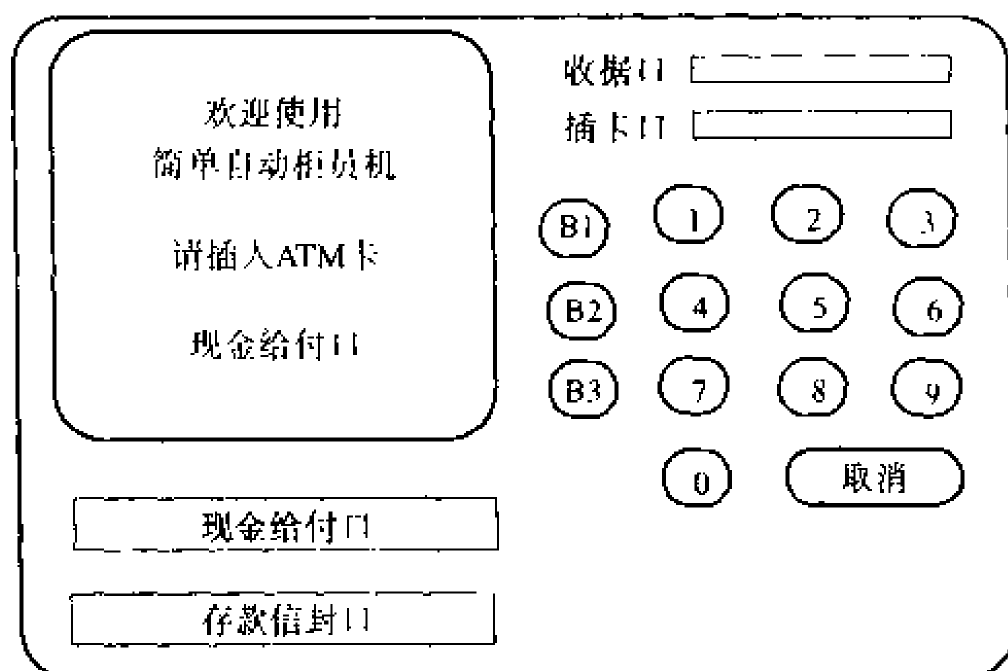


图12-7 SATM终端

这里采用如图12-8和图12-9所示的传统结构化分析方法描述SATM系统。这个模型是不完整的，但是包含了可说明所讨论测试技术的足够细节。需求规格说明的结构化分析方法现在仍然被广泛使用，并有大量CASE工具支持，像商业化的培训一样，并用文字进行描述。这种手段以三种互补模型为基础：功能、数据和控制。这里的功能模型使用数据流图，数据模型使用实体/关系模型，SATM系统的控制问题使用有限状态机。功能和数据模型采用Sybase公司的Deft CASE工具画出。这种工具用小写字母表示外部设备（例如终端通道），功能分解的元素用数字来标识（例如1.5表示“检验卡”功能）。流向的空心和实心箭头表示流动项是简单的还是复合的。这里给出的SATM系统部分一般属于系统的个人标识编号（PIN）检验部分。

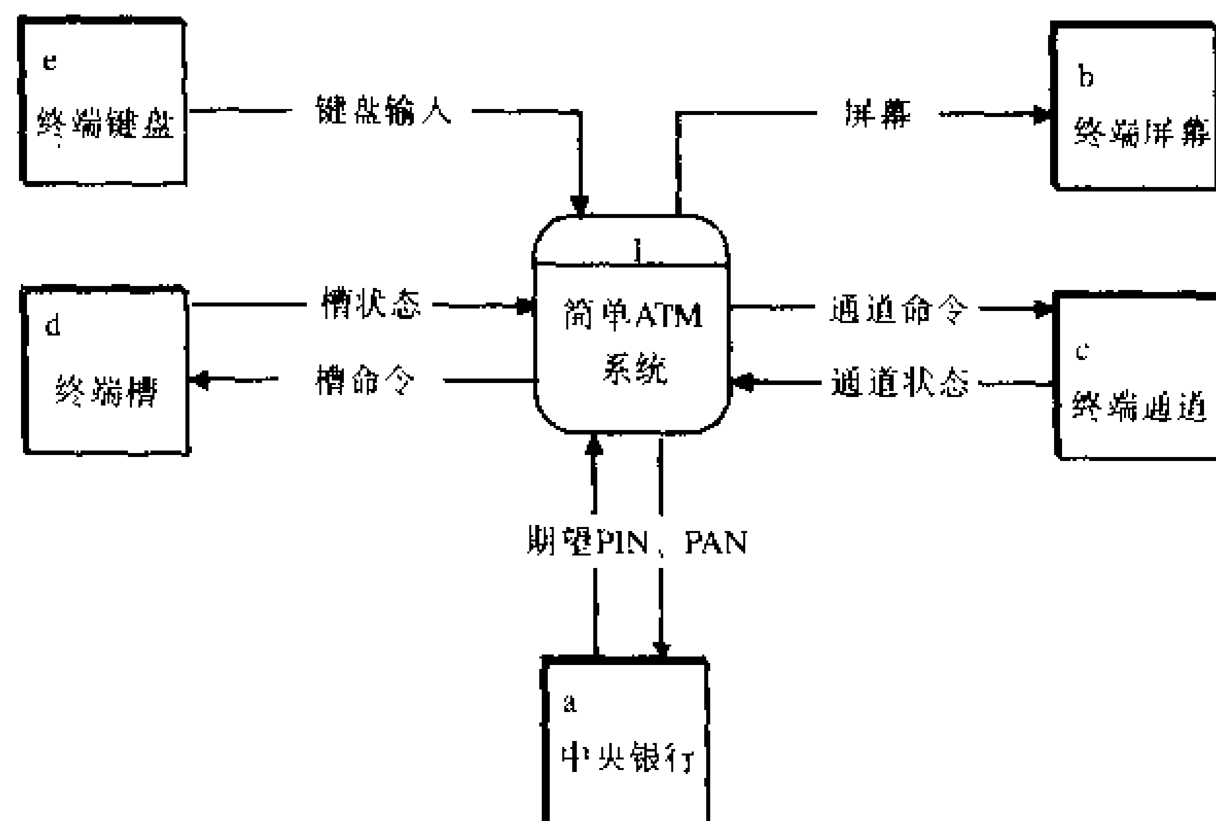


图12-8 SATM系统的语境图

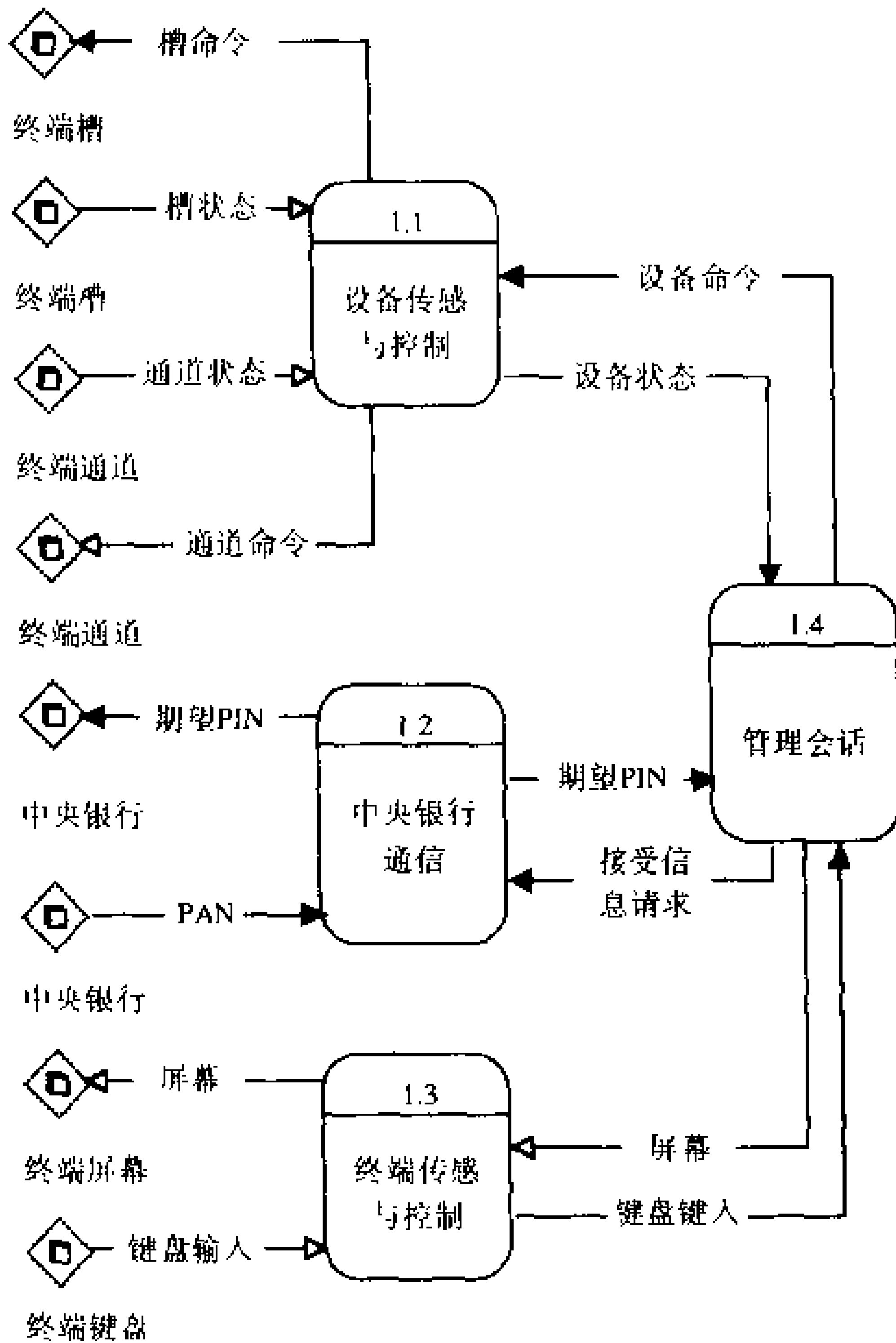


图12-9 SATM系统的第1层数据流图

Deft CASE区分简单和复合流，复合流可以分解为其他流，也可以是复合流，图形表示的简单流有实心箭头，复合流有空心箭头。

例如，复合流窗口有以下分解：

屏幕由以下内容组成

屏幕1	欢迎
屏幕2	输入PIN
屏幕3	PIN错误
屏幕4	PIN失败，退还ATM卡
屏幕5	选择事务处理类型
屏幕6	选择账户类型
屏幕7	输入账号

屏幕8　　资金不够
屏幕9　　不能取出此金额
屏幕10　　不能处理提取
屏幕11　　请取现金
屏幕12　　不能处理存入
屏幕13　　请将存款信封送入槽内
屏幕14　　需要其他事务处理吗？
屏幕15　　谢谢，请拿好ATM卡和收据

图12-10是SATM系统主要数据结构的一个（不完备的）实体/关系框图：客户、账户、终端和事务。好的建模实践要求为所保持数据描述的系统的每个部分设定一个实体（并被功能组件使用）。系统需要客户数据，包括每位客户的标识和个人账户编号（PAN）。这些信息已经编码到客户ATM卡的磁条中。还需要知道有关客户账户的信息，包括账号、余额、账户类型（储蓄账户还是支票账户）和账户的个人标识编号（PIN）。这时有人可能会问为什么PIN不与客户关联，PAN不与账户关联。规格说明中有这样的设计要求：如果询问这类数据，ATM卡有可能被任何人使用，因此这样的分离可预先执行安全检查过程。实体/关系模型部分描述实体之间的关系：客户“拥有”账户，客户在“会话”中引导事务处理，客户信息的独立性、事务处理“发生”在ATM终端上。单箭头和双箭头表示有一个或多个这样的关系：一个客户可以有多个账户、可以不引导或引导多个事务。多个事务处理可以发生在一台终端上，但是一个事务处理不会出现在多台终端上。

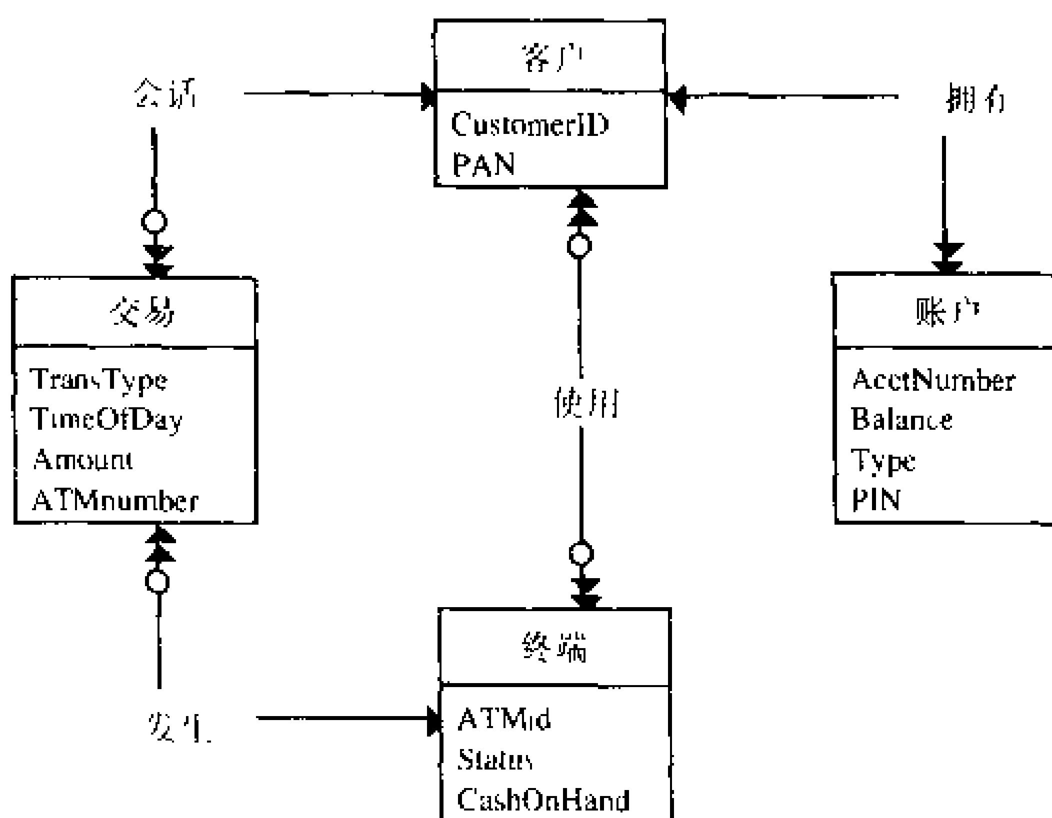


图12-10　SATM系统的实体/关系模型

数据流图和实体/关系模型包含主要结构的信息。这对于测试人员来说是有问题的，因

为测试用例关心的是行为，而不是结构。作为一种补充，功能和数据信息由控制模型连接到一起，这里我们使用有限状态机。控制模型表示结构和行为交叉点，因此对于测试人员特别有用。

如图12-11所示的上层有限状态机，将系统分为对应于客户使用阶段的状态。也有其他划分方法，例如可以根据所显示的屏幕选择状态（结果证明这种选择不好）。有限状态机可以以与数据流图很相似的方式分层分解。图12-12给出的是“等待PIN”状态的分解。在两张图中，状态的转移要么由ATM终端上的事件引起（例如键盘输入），要么由数据条件（例如判断PIN无误）引起。如果发生转移，还可能发生相应的行动。我们决定使用屏幕显示作为这类行动，实践证明在开发系统级测试用例时，这种选择非常方便。

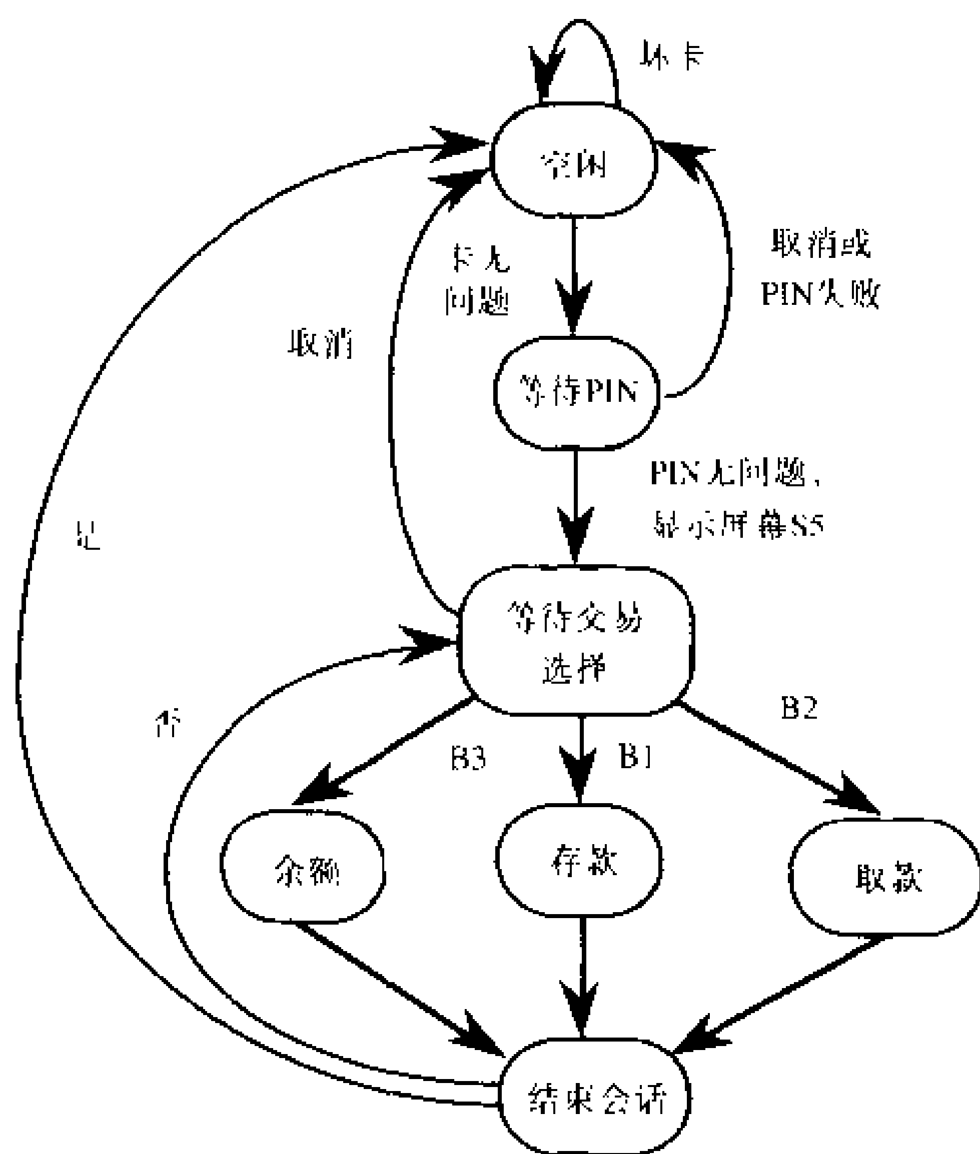


图12-11 上层SATM有限状态机

功能、数据和控制模型，在瀑布模型（以及新模型）中是设计活动的基础。在设计期间，有些最初的决定可能会根据额外的认识和更详细的需求（例如性能或可靠性目标）进行修改。最终结果是功能分解，例如如图12-13所示结构图中的一部分。请注意，最初的第一层分解的八个子系统已经不可见：功能已经被再分配给四个逻辑组件。做出这类选择是设计的基础，设计问题已经超出本书的范畴。在实践中，测试人员常常要忍受差的设计选择的结果

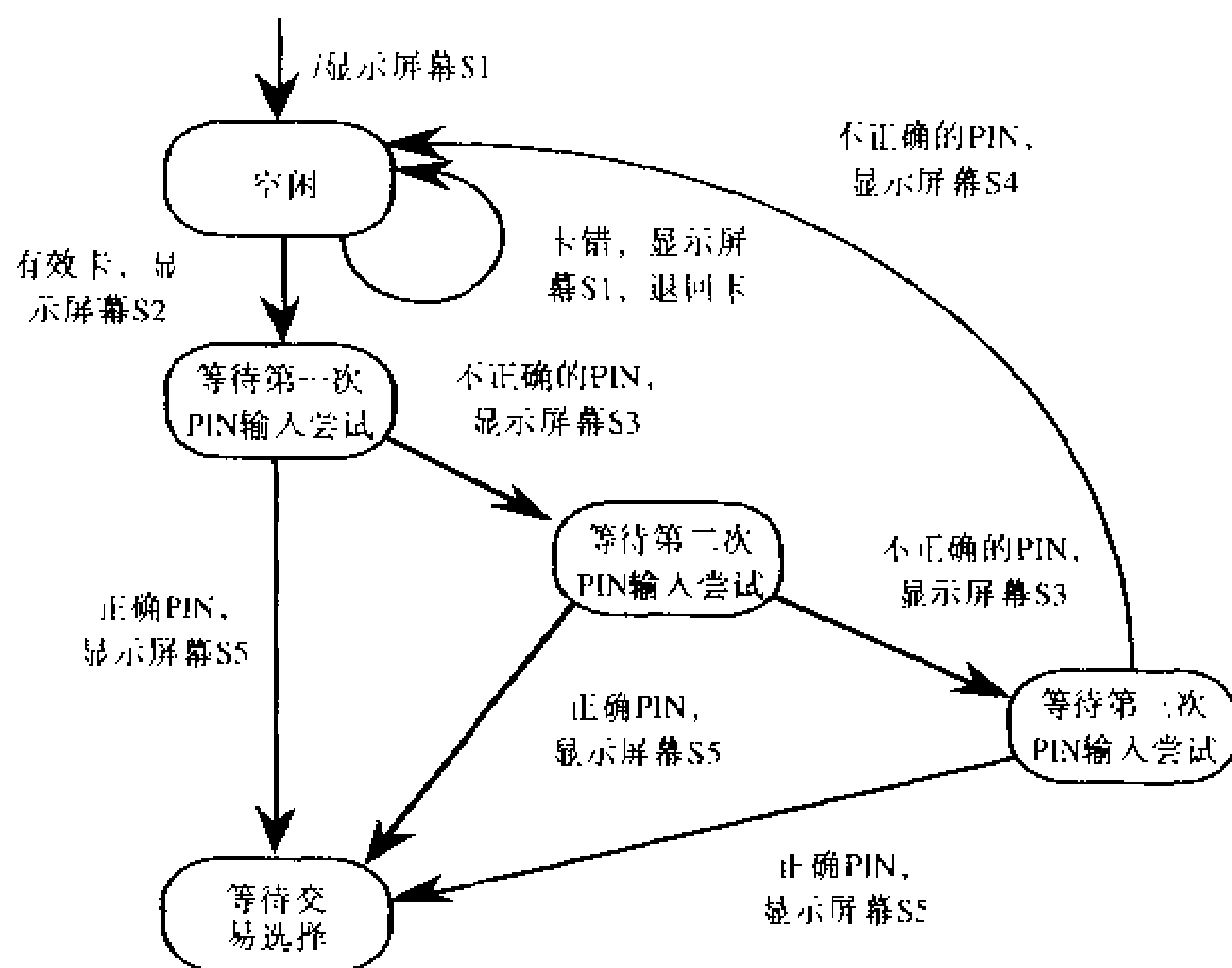


图12-12　PIN输入有限状态机

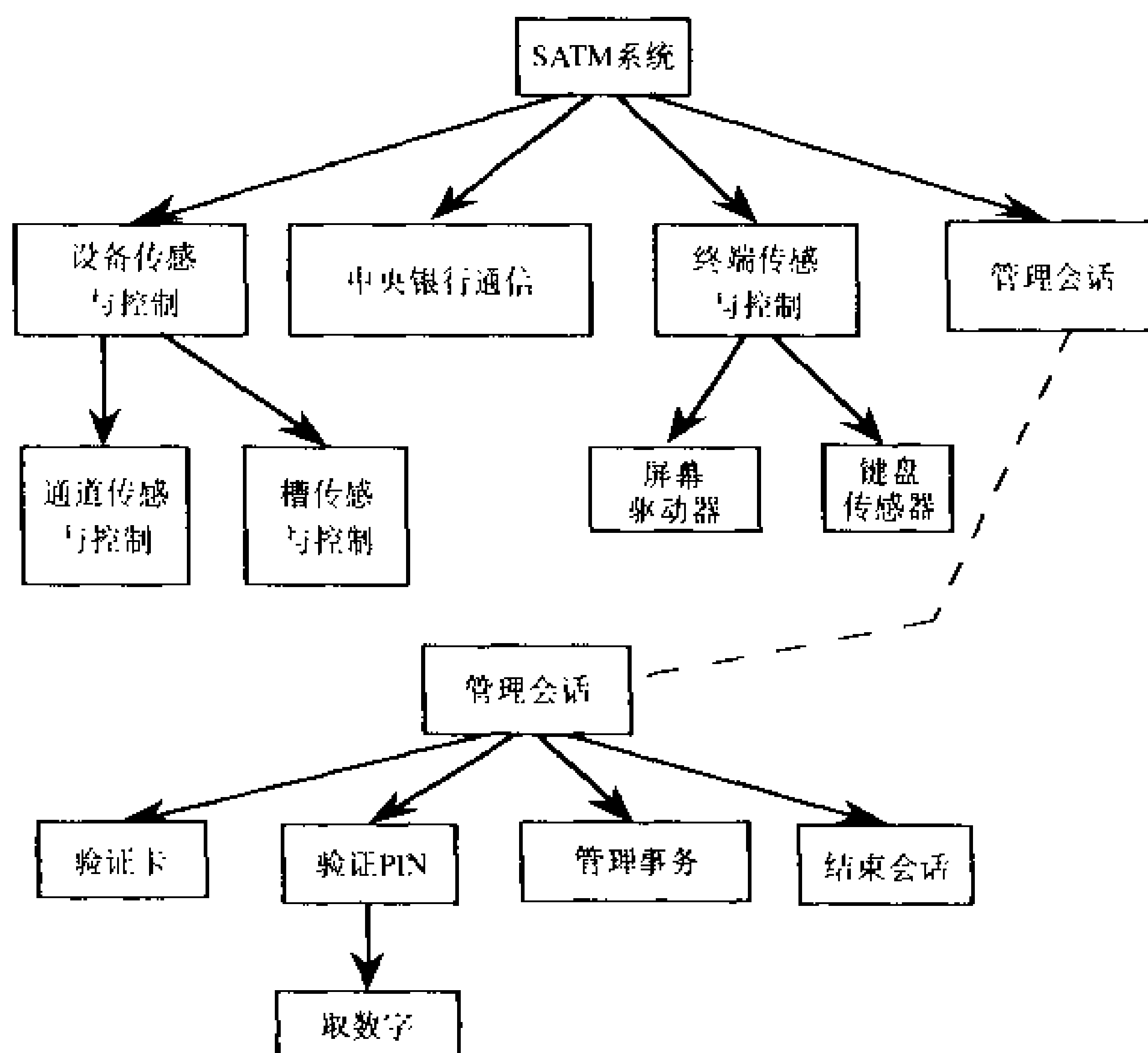


图12-13　SATM系统的一种分解树

如果只使用结构图指导集成测试，则会忽略有些（一般是低层）功能会用于多处这种事实。例如，这里的ScreenDriver函数在多个模块中使用，但是在功能分解中只出现一次。下一章将看到“调用图”是集成测试用例标识的一种好得多的基础。可以通过系统部分的更详细的视图开发这种调用图的起点。为了支持这种工作，需要带编号的功能分解以及两个组件的更详细的视图。

以下是功能分解，按照提纲形式排列：编号模式可反映如图12-13所示的组件层次

1 SATM System（SATM系统）
1.1 Device Sense & Control（设备功能与控制）
1.1.1 Door Sense & Control（通道功能与控制）
1.1.1.1 Get Door Status（取通道状态）
1.1.1.2 Control Door（控制通道）
1.1.1.3 Dispense Cash（给付现金）
1.1.2 Slot Sense & Control（槽功能与控制）
1.1.2.1 WatchCardSlot（检查ATM卡槽）
1.1.2.2 Get Deposit Slot Status（取存款槽状态）
1.1.2.3 Control Gard Roller（控制ATM卡传送器）
1.1.2.4 Control Envelope Roller（控制信封传送器）
1.1.2.5 Read Card Strip（读ATM卡磁条）
1.2 Central Bank Comm.（中央银行通信）
1.2.1 Get PIN for PAN（取PAN的PIN）
1.2.2 Get Account Status（取账户状态）
1.2.3 Post Daily Transactions（发出每日事务）
1.3 Terminal Sense & Control（终端功能与控制）
1.3.1 Screen Driver（屏幕驱动器）
1.3.2 Key Sensor（键盘传感器）
1.4 Manage Session（管理会话）
1.4.1 Validate Card（检验ATM卡）
1.4.2 Validate PIN（检验PIN）
1.4.2.1 GetPIN（取PIN）
1.4.3 Close Session（关闭会话）
1.4.3.1 New Transaction Request（新事务处理请求）
1.4.3.2 Print Receipt（打印收据）
1.4.3.3 Post Transaction Local（发送本地事务处理）
1.4.4 Manage Transaction（管理事务处理）
1.4.4.1 Get Transaction Type（取事务处理类型）

1.4.4.2　Get Account Type（取账户类型）

1.4.4.3　Report Balance（报告余额）

1.4.4.4　Process Deposit（处理存款）

1.4.4.5　Process Withdrawal（处理取款）

作为规格说明和设计过程的一部分，每个功能组件一般都要扩展，以显示输入、输出和处理机制。我们通过三个模块的伪代码（或PDL，程序设计语言）实现。主程序描述符合如图12-11所示有限状态机的描述。该框图中的状态用CASE语句实现。

```
Main Program
State = AwaitCard
Case State
Case 1: AwaitCard
        ScreenDriver(1, null)
        WatchCardSlot(CardSlotStatus)
        Do While CardSlotStatus is Idle
            WatchCardSlot(CardSlotStatus)
        End While
        ControlCardRoller(accept)
        ValidateCard(CardOK, PAN)
        If CardOK
            Then State = AwaitPIN
            Else    ControlCardRoller(eject)
        EndIf
        State = AwaitCard
Case 2: AwaitPIN
        ValidatePIN(PINok, PAN)
        If PINok
            Then    ScreenDriver(2, null)
                    State = AwaitTrans
            Else    ScreenDriver(4, null)
        EndIf
        State = AwaitCard
Case 3: AwaitTrans
        ManageTransaction
        State = CloseSession
Case 4: CloseSession
        If NewTransactionRequest
            Then    State = AwaitTrans
            Else    PrintReceipt
        EndIf
        PostTransactionLocal
        CloseSession
        ControlCardRoller(eject)
        State = AwaitCard
End Case (State)
End. (Main program SATM)
```

过程ValidatePIN基于如图12-12所示的有限状态机，其中的状态指尝试PIN输入的次数

```
Procedure ValidatePIN(PINok, PAN)
GetPINforPAN(PAN, ExpectedPIN)
```

```
Try = First
Case Try of
Case 1: First
        ScreenDriver(2, null)
        GetPIN(EnteredPIN)
        If EnteredPIN = ExpectedPIN
            Then    PINok = True
            Else    ScreenDriver(3, null)
        EndIf
        Try = Second
Case 2: Second
        ScreenDriver(2, null)
        GetPIN(EnteredPIN)
        If EnteredPIN = ExpectedPIN
            Then    PINok = True
            Else    ScreenDriver(3, null)
        EndIf
        Try = Third
Case 3: Third
        ScreenDriver(2, null)
        GetPIN(EnteredPIN)
        If EnteredPIN = ExpectedPIN
            Then    PINok = True
            Else    ScreenDriver(4, null)
                    PINok = False
        EndIf
EndCase (Try)
End.   (Procedure ValidatePIN)
```

过程GetPIN基于另一个有限状态机，其中的状态指接收到的数字个数，在任何状态中，要么键入另一个能够键入的数字键，要么键入能够键入的取消键。这些“状态”没有再使用CASE语句实现，而是压缩为while循环迭代处理。

```
Procedure GetPIN(EnteredPIN, CancelHit)
Local Data: DigitKeys = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

CancelHit = False
EnteredPIN = null string
DigitsRcvd=0
Do While NOT(DigitsRcvd=4 OR CancelHit)
    KeySensor(KeyHit)
    If KeyHit IN DigitKeys
        Then
            EnteredPIN = EnteredPIN + KeyHit
            INCREMENT(DigitsRcvd)
            If DigitsRcvd=1 Then ScreenDriver(2,'X---')
            If DigitsRcvd=2 Then ScreenDriver(2,'XX--')
            If DigitsRcvd=3 Then ScreenDriver(2,'XXX-')
            If DigitsRcvd=4 Then ScreenDriver(2,'XXXX')
        Else
            CancelHit = True
        EndIf
  End While
End.   (Procedure GetPIN)
```

如果仔细研究这三个模块的伪代码，就可以找出功能分解中模块之间的“使用”关系。第13章将讨论这种关系如何提供对集成测试的认识。

模块	使用模块
SATM主程序	WatchCardSlot（检查ATM卡槽）
	Control Card Roller（控制ATM卡传送器）
	Screen Driver（屏幕驱动器）
	Validate Card（检验ATM卡）
	Validate PIN（检验PIN）
	Manage Transaction（管理事务处理）
	New Transaction Request（新事务处理请求）
检验PIN	GetPINforPAN（取PAN的PIN）
	GetPIN（取PIN）
	Screen Driver（屏幕驱动器）
取PIN	KeySensor（键盘传感器）
	Screen Driver（屏幕驱动器）

请注意，“使用”信息并不会现成地出现在功能分解中，而是在设计过程的更详细阶段开发出来的。第13章还会讨论这个问题。

12.4 将集成测试与系统测试分开

我们现在几乎能够清晰地区分集成测试和系统测试了。这种区分能够避免不同层次测试之间的漏洞和冗余，澄清这些层次的适当测试目标，理解在不同层次上如何标识测试用例。本节所利用的概念所有讨论对于所有层次的测试都是最基本的，即“线索”概念。线索是与执行时间行为有关的构造。当测试系统时，我们要使用测试用例选择（并执行）线索。我们可以对线索分层：系统线索描述系统级行为，集成线索对应集成级行为，单元线索对应单元级行为。很多文献都使用了这个术语，但是很少给出定义。而在给出定义的文献中，所给出的定义也没有多少帮助。现在，我们把“线索”当做原语，很像函数和数据。第14章将给出线索很具体的定义。在下两章中，我们将会看到线索常常通过系统的描述和开发方式识别。例如，可以把线索看做是通过系统的有限状态机描述的一条路径，或看做是由数据语境和端口级输入事件序列确定的内容，例如SATM系统背景图中的内容，或看做是机器指令序列。重要的是，线索是一般概念，独立于系统描述和开发的方式而存在。

我们已经观察了结构视图和行为视图的差别，这两种视图会有助于区分集成测试和系统测试。结构视图通过构建系统的过程和构建系统所使用的技术，反映两种测试的过程。我们当然期望不同层次的测试用例可以反向跟踪到开发信息上。尽管这是必要的，但却不是充分的：最终还是要通过行为构造作所需的分离。

12.4.1 结构认识

所有人都同意必须进行区分，而且集成测试所处的层次要比系统测试细得多。此外大家还普遍认为，集成测试可以安全地假设单元已经通过单独测试，并且单元个体已经能够正常起作用。因此，我们都认为集成测试要考虑的是单元之间的接口。

一种可能是回到瀑布生命周期模型中的对称结构上，认为集成测试考虑的是概要设计信息，而系统测试是在需求规格说明的层次上。这是一种很流行的学术观点，但是有一个重要问题：我们怎样区分规格说明和概要设计？学术界对这个问题恰当的回答是，什么与如何之间的差别：需求规格说明定义是什么，概要设计定义怎样做。尽管初看起来这种回答不错，但是在实践中行不通。有些学者认为，甚至需求规格说明技术的选择也是设计选择。

那些常常对需求规格说明抱有“别限制我”态度的设计人员对生命周期方法的回应是：需求规格说明既不应该指定也不应该排除设计选择。根据这种观点，如果规格说明中的信息详细到“激怒设计人员”的程度，规格说明就太详细了。这听起来很好，但是仍然不能产生区分集成测试和系统测试的方法。

开发过程中所使用的模块可提供一些线索。如果仔细研究SATM系统的定义，就会首先要求系统测试要保证生成所有15个显示屏幕（一种基于输出定义域的系统测试功能视图）实体/关系模型也有帮助：一对一和一对多关系有助于我们理解必须完成多大量的测试。控制模型（在这个例子中是有限状态机层次结构）最有帮助，我们可以通过有限状态机路径的方式对系统测试用例提出假设，这样做会产生一种系统级的结构性测试。功能模型（数据流图和结构图表）在层次方向上移动，因为两种模型都描述功能分解。即使这样，我们也不能通过结构图表确定在什么地方系统测试停止、集成测试开始。我们利用结构信息能够做的是标识极值。例如，以下线索在系统层上都是清晰的：

1. 插入无效ATM卡（这可能是“最短”的系统线索）。
2. 插入有效ATM卡，但是后面的三次PIN输入尝试都失败。
3. 插入有效ATM卡，PIN输入尝试正确，然后是余额查询。
4. 插入有效ATM卡，PIN输入尝试正确，然后是存款。
5. 插入有效ATM卡，PIN输入尝试正确，然后是取款。
6. 插入有效ATM卡，PIN输入尝试正确，然后是试图提取比账户余额多的现金

我们还可以找出一些集成级线索。下面回到ValidatePIN和GetPIN伪代码描述ValidatePIN调用GetPIN，GetPIN等待KeySensor报告所键入的键，如果键入的是数字，则GetPIN在显示屏幕上会显示一个“X”；如果输入的是取消键，则GetPIN终止，ValidatePIN考虑另一次PIN输入尝试。我们还可以再向下走一点，考虑键入顺序，例如两三个数字后输入取消键。

12.4.2 行为认识

在业界应用中，有一种有实效的显式区分效果不错。以系统的系统级输入、输出位置上的端口边界来考虑系统。每个系统都有端口边界。SATM系统的端口边界包括数字小键盘、功能键、屏幕、存款和取款通道、进出ATM卡和收据槽等。客户可以看到端口输入和输出事件，客户往往通过一系列端口事件理解系统行为。因此，我们要求系统端口事件为系统测试用例的“原语”，即系统测试用例（或等价的系统线索）采用端口输入和端口输出事件的交替序列来描述。这符合我们对测试用例的理解，对于测试用例我们要描述前提、输入、输出和后果。通过这种要求，我们总能够发现层次冲突：如果测试用例（线索）请求在端口边界不可见的输入（或输出），则该测试用例就不能是系统级测试用例（线索）。请注意，这是清晰、可识别和可实施的。第14章在讨论系统行为线索时还会详细研究这个问题。

线索支持测试的高度解析的视图。例如单元级线索是执行（可行路径的）源语句的序列，集成级线索可以被看做是单元级线索序列，但不考虑单元线索的“内部问题”，只考虑单元线索的交互。最后，系统级线索可以解释为集成级线索序列。我们还能够描述系统级线索之间的交互。为了对比两种测试，以下两章将把这些线索捆绑在一起

12.5 参考文献

Agresti, W.W., *New Paradigms for Software Development,* IEEE Computer Society Press, Washington, D.C., 1986.

Boehm, B.W., A spiral model for software development and enhancement, *IEEE Computer,* Vol. 21, No. 6, IEEE Computer Society Press, Washington, D.C., May 1988, pp. 61–72

Topper, Andrew et al., *Structured Methods: Merging Models, Techniques, and CASE,* McGraw-Hill, New York, 1993.

第13章

集成测试

1999年9月，火星气象轨道人造卫星的使命，在经过41周4.16亿英里的成功飞行之后，终于失败了。这颗卫星在就要开始进入火星轨道时消失了。卫星的缺陷本来可以通过集成测试查出：洛克希德·马丁太空科学家使用的是英制（磅）加速度数据，而喷气推进实验室采用公制（牛顿）加速度数据进行计算。NASA宣布了一项5万美元的计划调查出现这种问题的原因（Fordahl，1999）。他们本来应该阅读本章。

工艺师有两个基本特征：他们对自己行业的工具有很深的理解，对为之工作的媒介也有相似的理解，以将自己对工具的理解反映到如何使用工具加工媒介上。本书第二部分和第三部分集中讨论了测试工艺师所拥有的工具（技术）上。我们的目标是通过工具关于特定软件类型的优点和局限性来理解这些测试技术。现在，我们的关注重点要转向媒介，目标是通过更好地理解媒介来提高测试工艺师的判断能力。这里我们特意做以下区分：本章和下一章将讨论采用传统功能、数据、控制和结构模型来定义、设计和开发的软件测试问题。面向对象软件的测试问题在第五部分讨论。本章继续使用改进版简单自动柜员机（SATM）系统的例子，说明集成测试的三种不同方法。对于每种方法，我们首先讨论基本知识，然后讨论使用这些基本信息的各种手段。为了继续对工艺师进行类比，我们强调每种集成测试技术的优点和局限性。

13.1 深入研究SATM系统

第12章通过输出屏幕（如图12-6所示）、实际终端（如图12-7所示）、系统语境图和部分数据流图（如图12-8和12-9所示）、系统数据的实体/关系模型（如图12-10所示）、描述部分系统行为的有限状态机（如图12-11和12-12所示），以及部分功能分解（如图12-13），描述了SATM系统，还开发了主程序和两个单元ValidatePIN和GetPIN的伪代码描述。

这里首先扩展在图12-12中开始的功能分解，其中的编号模式反映了图中组件的层次。为了便于引用，在以下分析中出现的每个组件，都给出一个新（更短的）编号，这些编号在表13-1中给出。（这样做的惟一理由是使数字更加可读。）

表13-1 SATM单元和缩写名称

单元编号	层次编号	单元名称
1	1	SATM System（SATM系统）
A	1.1	Device Sense & Control（设备功能与控制）
D	1.1.1	Door Sense & Control（通道功能与控制）
2	1.1.1.1	Get Door Status（取通道状态）
3	1.1.1.2	Control Door（控制通道）
4	1.1.1.3	Dispense Cash（给付现金）
E	1.1.2	Slot Sense & Control（槽功能与控制）
5	1.1.2.1	WatchCardSlot（检查ATM卡槽）
6	1.1.2.2	Get Deposit Slot Status（取存款槽状态）
7	1.1.2.3	Control Card Roller（控制ATM卡传送器）
8	1.1.2.4	Control Envelope Roller（控制信封传送器）
9	1.1.2.5	Read Card Strip（读ATM卡磁条）
10	1.2	Central Bank Comm.（中央银行通信）
11	1.2.1	Get PIN for PAN（取PAN的PIN）
12	1.2.2	Get Account Status（取账户状态）
13	1.2.3	Post Daily Transactions（发出每日事务处理）
B	1.3	Terminal Sense & Control（终端功能与控制）
14	1.3.1	Screen Driver（屏幕驱动器）
15	1.3.2	Key Sensor（键盘传感器）
C	1.4	Manage Session（管理会话）
16	1.4.1	Validate Card（检验ATM卡）
17	1.4.2	Validate PIN（检验PIN）
18	1.4.2.1	GetPIN（取PIN）
F	1.4.3	Close Session（关闭会话）
19	1.4.3.1	New Transaction Request（新事务处理请求）
20	1.4.3.2	Print Receipt（打印收据）
21	1.4.3.3	Post Transaction Local（发送本地事务处理）
22	1.4.4	Manage Transaction（管理事务处理）
23	1.4.4.1	Get Transaction Type（取事务处理类型）
24	1.4.4.2	Get Account Type（取账户类型）
25	1.4.4.3	Report Balance（报告余额）
26	1.4.4.4	Process Deposit（处理存款）
27	1.4.4.5	Process Withdrawal（处理取款）

表13-1给出的分解可表示为如图13-1所示的分解树。这种分解是集成测试常用视图的基础。重要的是要注意这种分解主要是系统的打包划分。随着软件设计逐步深入，所增加的信息能够用来将功能分解树细化为单元调用图。单元调用图是一种有向图，节点表示程序单元，边对应程序调用。即，如果单元A调用单元B，则从单元A到单元B有一条有向边。第12章在研究主程序和ValidatePIN和GetPIN模块的调用时，我们已经开始开发SATM系统的调用图了。所得到的信息由表13-2中的邻接矩阵表示。这个矩阵采用电子表格创建，实践证明电子表格是测试人员非常好用的工具。

表13-2 SATM调用图的邻接矩阵

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
1				X		X							X		X	X		X	X	X	X					
2																										
3																										
4																										
5																										
6																										
7																										
8																										
9																										
10																										
11																										
12																										
13																										
14																										
15																										
16								X	X		X															
17										X			X				X									
18													X	X												
19													X	X												
20																										
21																						X	X	X	X	X
22													X	X												
23													X	X												
24													X	X												
25														X												
26	X	X			X		X					X	X	X												
27	X	X	X		X		X					X	X	X												

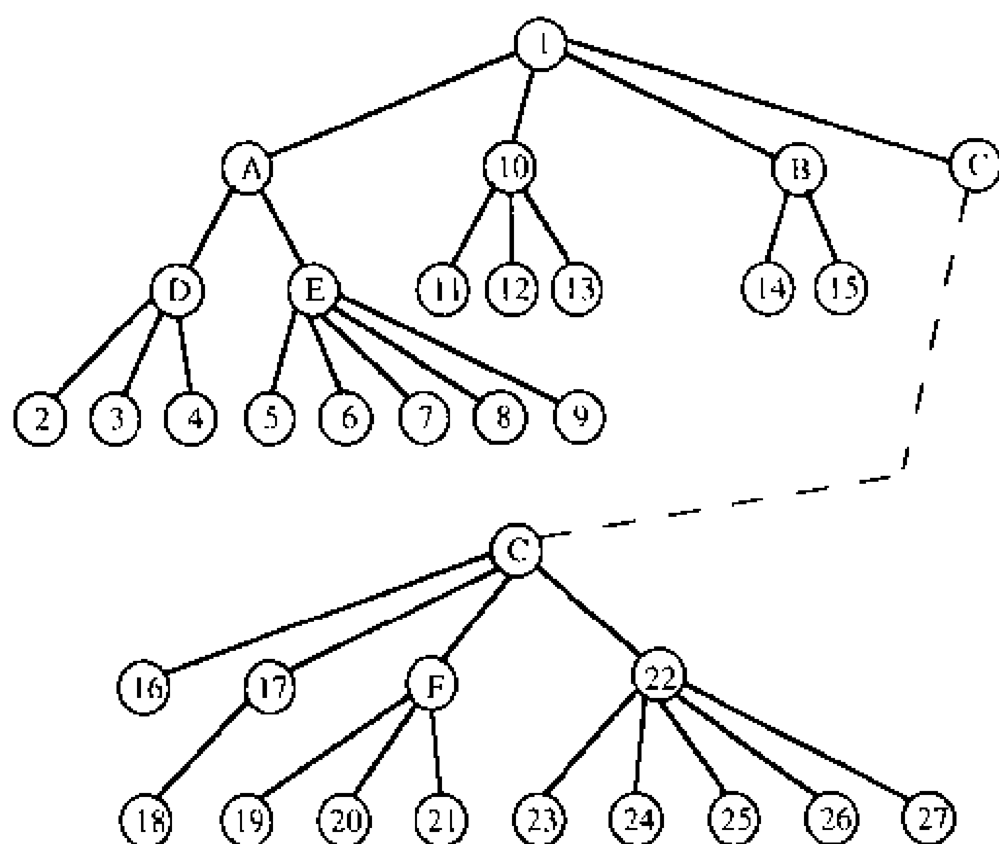


图13-1 SATM功能分解树

SATM调用图如图13-2所示。部分层次结构被隐藏起来以使图更清晰。有一点相当明显：调用图没有按比例给出。调用图和邻接矩阵都会加深测试人员的认识。对于内度和外度很高的节点，集成测试很重要，从主程序（节点1）到汇节点的路径可以用来标识增量开发的构建内容。

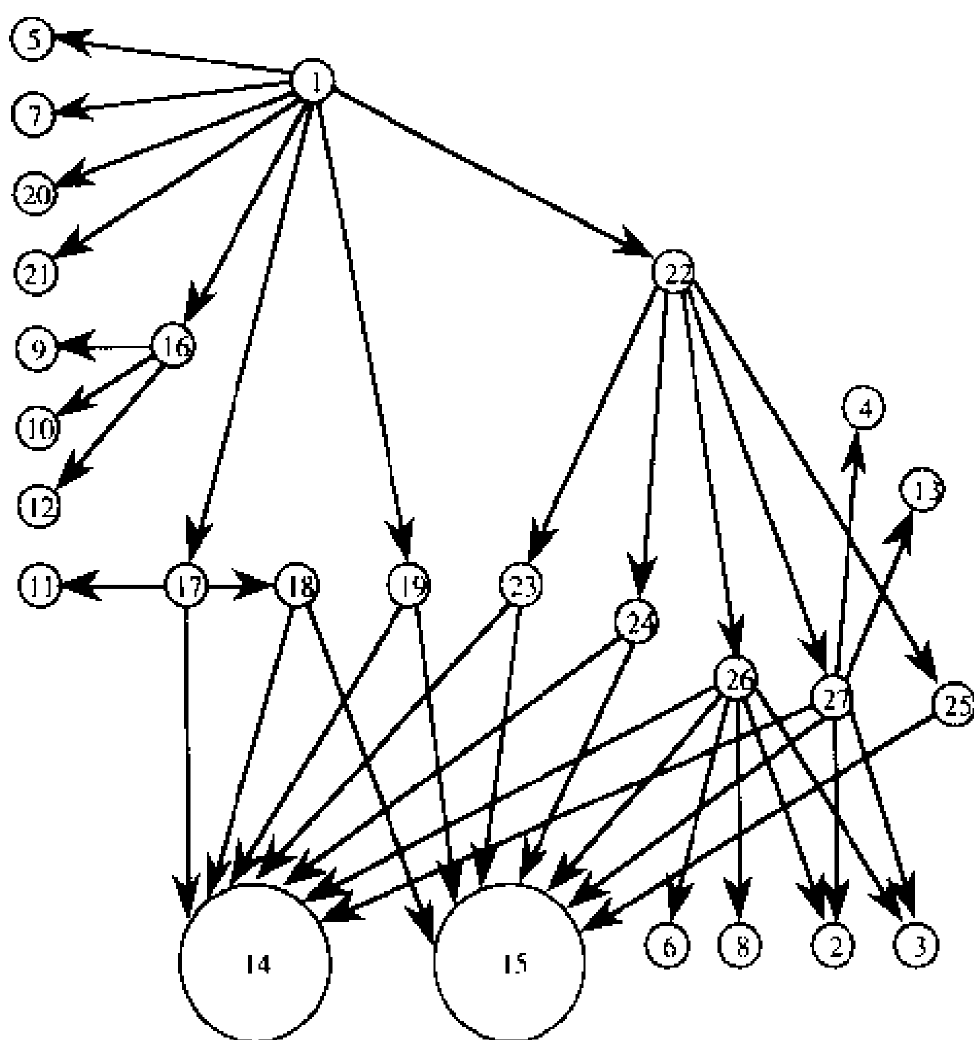


图13-2 SATM调用图

13.2 基于分解的集成

大多数教科书在讨论集成测试时，都只考虑基于系统测试功能分解的集成测试。这些方法都基于功能分解，要么采用树（如图13-1所示）表示，要么采用文字形式表示。这类讨论不可避免地要深入要集成的模块顺序。有四种选择：从树顶开始向下（自顶向下）、从树底开始向上（自底向上）、这两种方法的某种组合（三明治）或这些顺序以外的顺序（大爆炸）。所有这些集成顺序都假设单元已经通过单独测试，基于分解的集成的目标是测试通过单独测试的单元接口。

大爆炸方法最容易：这种集成将所有单元放在一起编译并进行一次性测试。这种方法的缺点是，当（不是如果！）发现失效时，没有多少线索能够用来帮助确定缺陷位置。（请注意第1章讨论过的缺陷和失效之间的差别。）

13.2.1 自顶向下集成

自顶向下集成从主程序（树根）开始，所有被主程序调用的下层单元都作为"桩"出现。桩就是模拟被调用单元的一次性代码。如果要为SATM系统执行自顶向下集成测试，第一步就应该是为被主程序调用的所有单元，即WatchCardSlot（检查ATM卡槽）、Control Card Roller（控制ATM卡传送器）、Screen Driver（屏幕驱动器）、Validate Card（检验ATM卡）、ValidatePIN（检验PIN）、Manage Transaction（管理事务处理）和New Transaction Request（新事务处理请求）开发桩。一般来说，测试人员不得不开发桩，并且还要做一定的集成。以下是两个桩的示例：

```
Procedure GetPINforPAN(PAN, ExpectedPIN) STUB
If PAN = '1123' Then PIN := '8876'
If PAN = '1234' Then PIN := '8765'
If PAN = '8746' Then PIN := '1253'
End

Procedure KeySensor(KeyHit) STUB
data: KeyStrokes STACK OF '8', '8', '7', 'cancel'
KeyHit = POP (KeyStrokes)
End
```

在GetPINforPAN（取PAN的PIN）的桩中，测试人员复制了一个有少量将出现在测试用例中的取值的查对表。在KeySensor（键盘传感器）的桩中，测试人员必须提供每次当KeySensor过程被调用时会出现的端口事件序列。（这里提供通过键盘输入的PIN "8876"，但是用户在第四个数字输入之前先键入取消键。）在实践中，开发桩的工作量常常很大。有很好的理由把桩代码看做是软件开发的一部分，并在配置管理下维护。

一旦提供了SATM主程序的所有桩之后，可开始测试主程序，就像它是一个独立单元一样，可以使用合适的功能性测试和结构性测试技术，并查找缺陷。在确信主程序的逻辑正确之后，我们将逐渐采用实际代码取代这些桩。图13-3给出了图13-1中SATM分解的部分自

顶向下集成测试序列。在最上层，要开发第一层分解中四个组件的桩。共有四个集成过程，在每个过程中，一个组件是实际（以前通过单元测试的）代码，其他三个是桩。自顶向下集成采用功能分解树的广度优先遍历策略。图13-3还给出了两个额外的集成层。

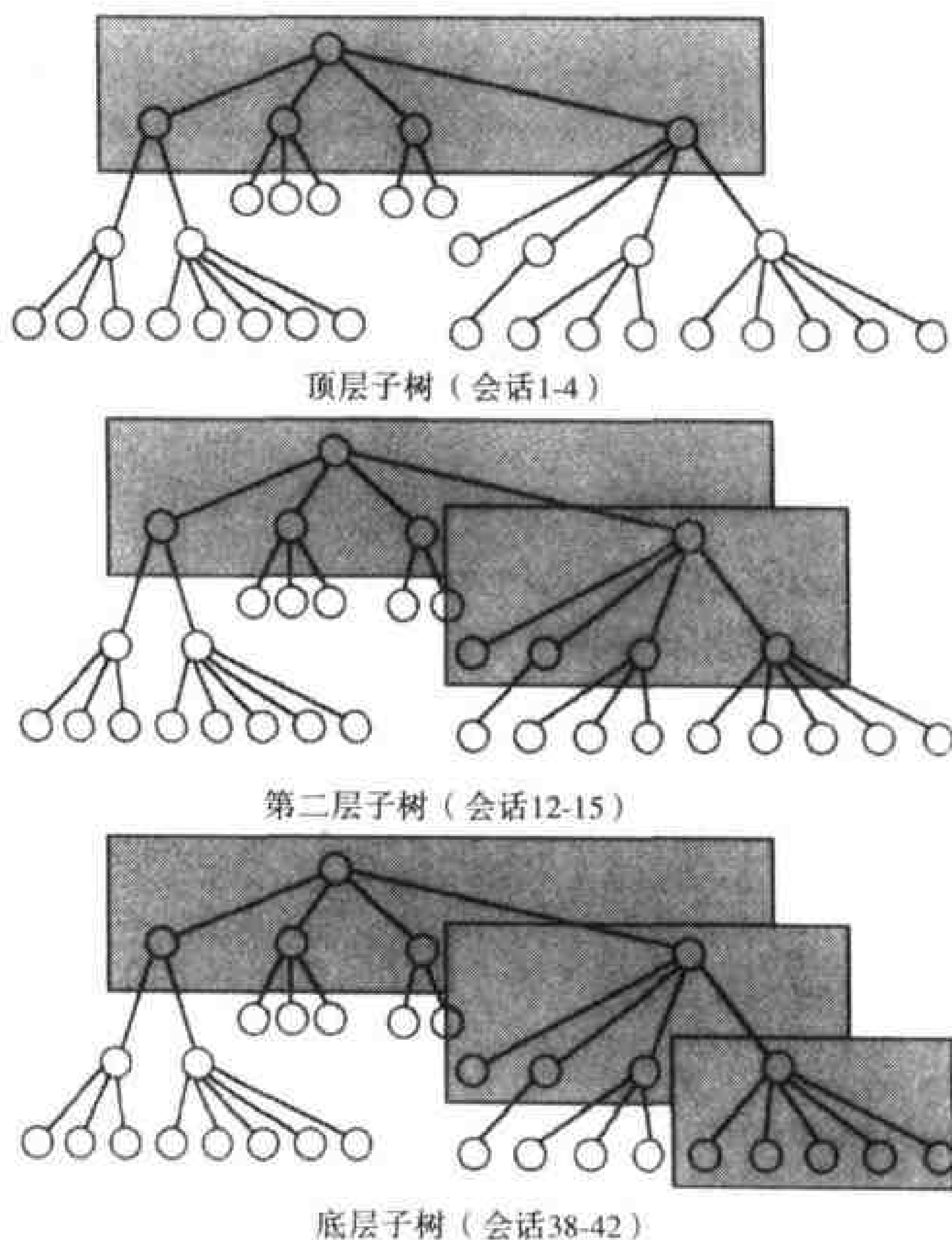


图13-3 自顶向下集成

即使这样做也是有问题的。需要一次替代所有桩吗？如果是这样，将会面临高外度单元的"小爆炸"。如果一次替代一个桩，则每替代一个桩都要重新测试一遍主程序。这意味着，对于SATM主程序这个例子，需要重复集成测试八次（每次替代一个桩要重新测试一次，一次是对所有桩的情况测试）。

13.2.2 自底向上集成

自底向上集成是自顶向下顺序的"镜像"，不同的是，桩由模拟功能分解树上一层单元的驱动器模块替代（如图13-4所示）。在自底向上集成中，首先从分解树的叶开始（像ControlDoor和DispenseCash这样的单元），并用特别编写的驱动器测试。驱动器中的一次性代码比桩中的少。大多数系统在接近叶时都有相当高的扇出数，因此在自底向上集成顺序中，不需要同样数量的驱动器，不过代价是驱动器模块都比较复杂。

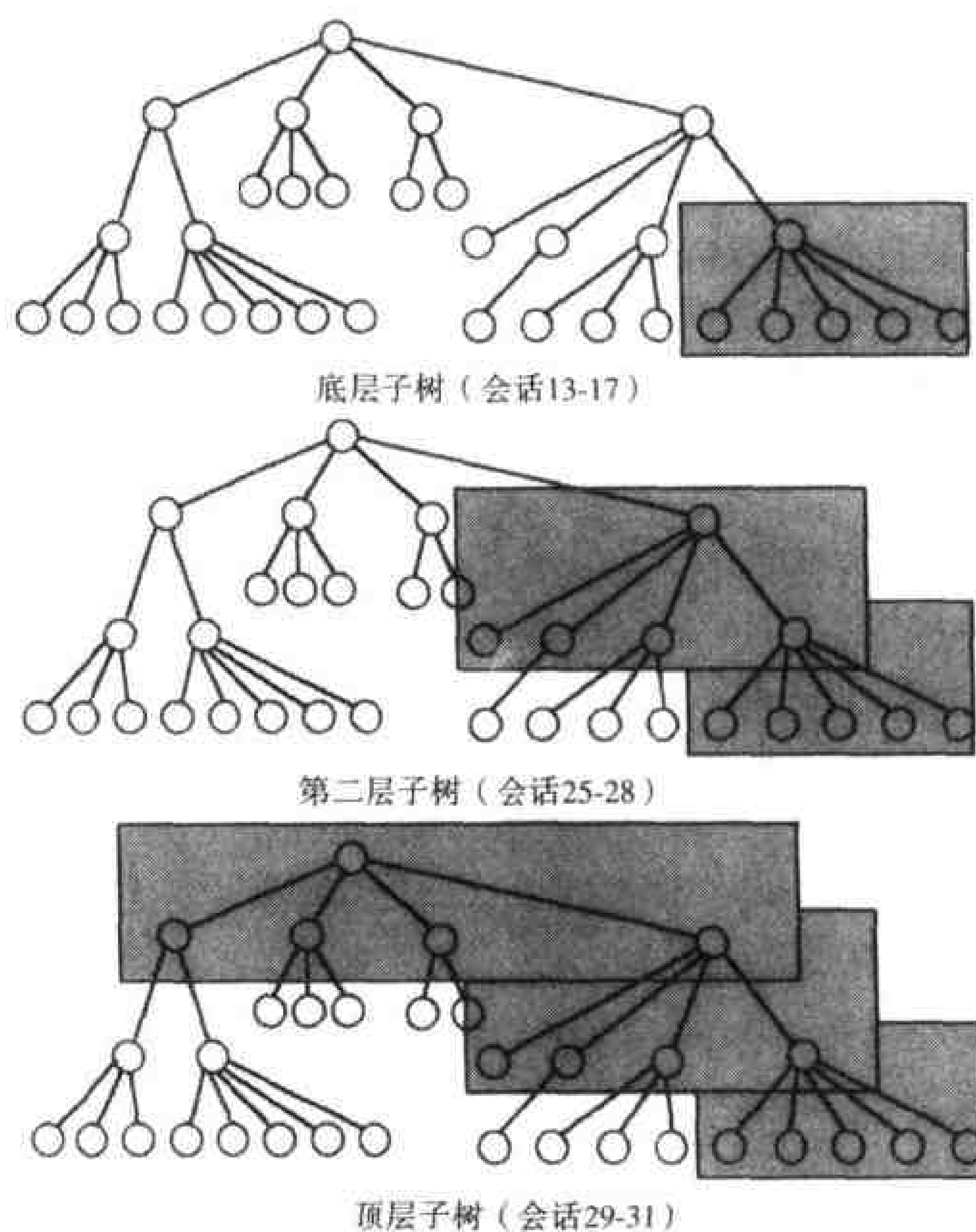


图13-4 自底向上集成

13.2.3 三明治集成

三明治集成是自顶向下和自底向上集成的组合。如果通过分解树考虑三明治集成，则只需要在子树上真正进行大爆炸集成（如图13-5所示）。桩和驱动器的开发工作都比较小，不过代价是作为大爆炸集成的后果，在一定程度上增加了定位缺陷的难度。（我们可能可以讨论三明治的尺寸，从美味手指三明治到多层三明治，不过现在先不讨论。）

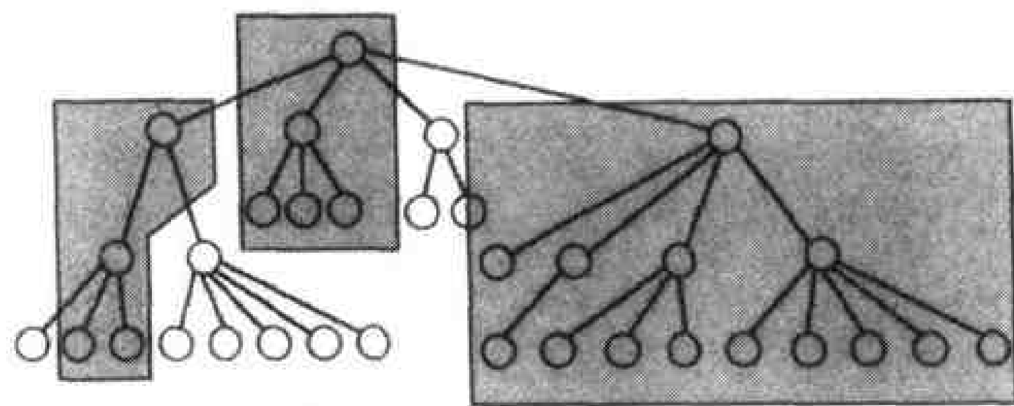


图13-5 三明治集成

13.2.4 优缺点

除了大爆炸集成，基于分解的方法在直觉上都很清晰，都用经过测试的组件构建。只要发现失效，就怀疑最新加入的单元。集成测试很容易根据分解树跟踪（如果分解树很小，随着节点被成功地集成，树逐渐变成节点）。自顶向下和自底向上采用广度优先的分解树遍历策略，但并不是必须这样做（我们可以使用全高度三明治，以深度优先的方式测试分解树。）

功能分解和瀑布开发经常遇到的非议之一是两者都是人工的，更多的还是为了满足项目管理的需要，而不是为了满足软件开发人员的需要。基于分解的测试也是这样。整个机制是根据结构集成单元，假设正确行为来自个体正确的单元和正确的接口（实践者对这一点理解得更好。）桩或驱动器的开发工作量是这些方法的另一个缺点。此外还有重新测试所需工作量的问题。以下是给定分解树所需集成测试会话数的计算公式（一个测试过程是为一个特定配置的实际代码和桩所做的一组测试）：

会话＝节点－页＋边

SATM系统有42个集成测试过程，这意味着42个单独的集成测试用例集合。

对于自顶向下集成，需要开发（节点－1个）桩；对于自底向上集成，需要开发（节点－叶个）驱动器。对于SATM系统，为32个桩和10个驱动器。

13.3 基于调用图的集成

基于分解集成的缺点之一是以功能分解树为基础。如果改用调用图，则可以减缓这种缺陷，并且也向结构性测试方向发展。现在可以利用前面介绍过的图论知识。由于调用图是一种有向图，为什么不像使用程序图那样地使用调用图呢？这种问题导致集成测试的两种新方法，我们把它们叫做成对集成和相邻集成。

13.3.1 成对集成

成对集成的思想是免除桩/驱动器开发工作。为什么要开发桩和/或驱动器，而不是使用实际代码呢？初看起来，这类似大爆炸集成，都是我们把其限制在调用图中的一对单元上。最终结果是对调用图中的每条边有一个集成测试会话（对于图13-2所示的SATM调用图，有40个集成测试过程），但是可以大大降低桩/驱动器的开发工作。图13-6给出了四个成对集成过程。

13.3.2 相邻集成

可以通过借用拓扑学中的相邻概念，让数学家把我们再推进一步（这一步不算太大，因为图论是拓扑学的一个分支）。图中节点的邻居，是边从给定节点引出的节点集合。在有向图中，节点邻居包括所有直接前驱节点和所有直接后继节点（请注意，这对应节点的桩和驱动器集合）。节点16和26的邻居如图13-7所示，SATM例子的11个邻居（根据图13-2给出的调用图）在表13-3中列出。

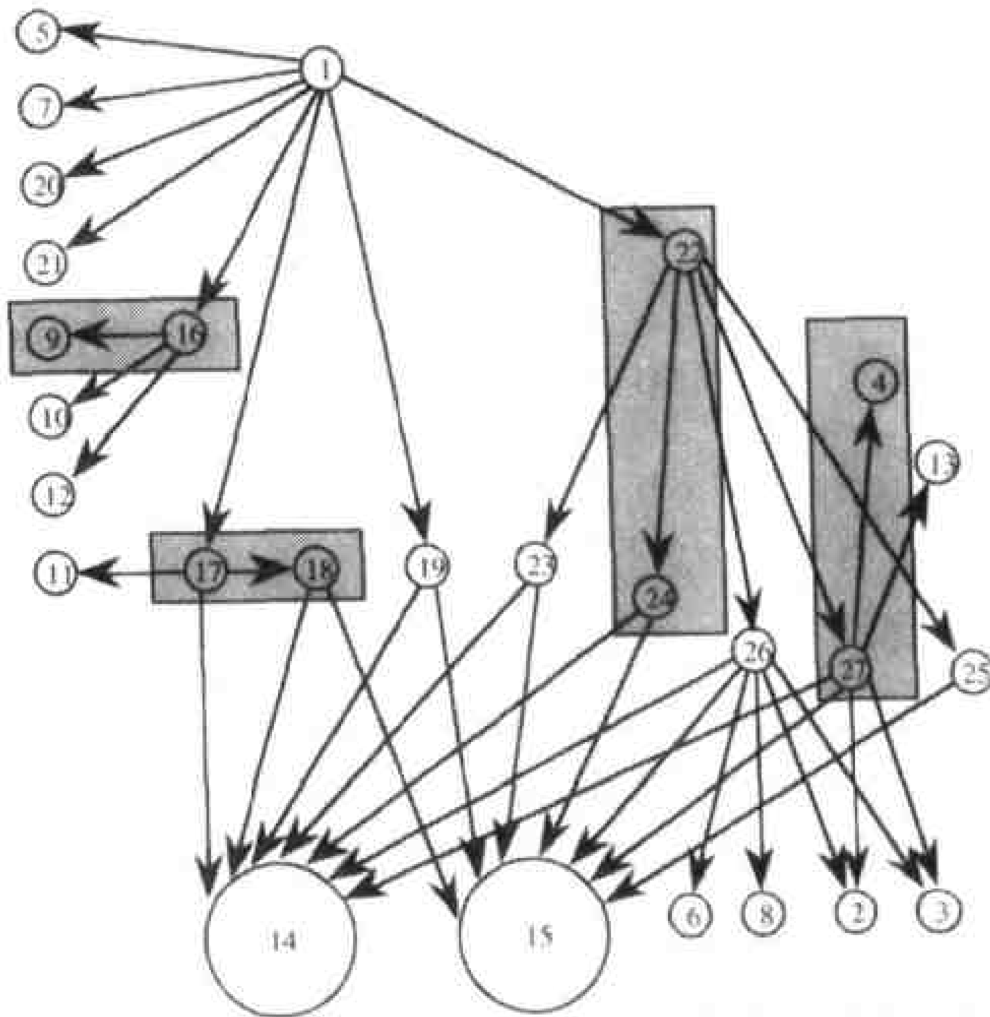


图13-6 成对集成

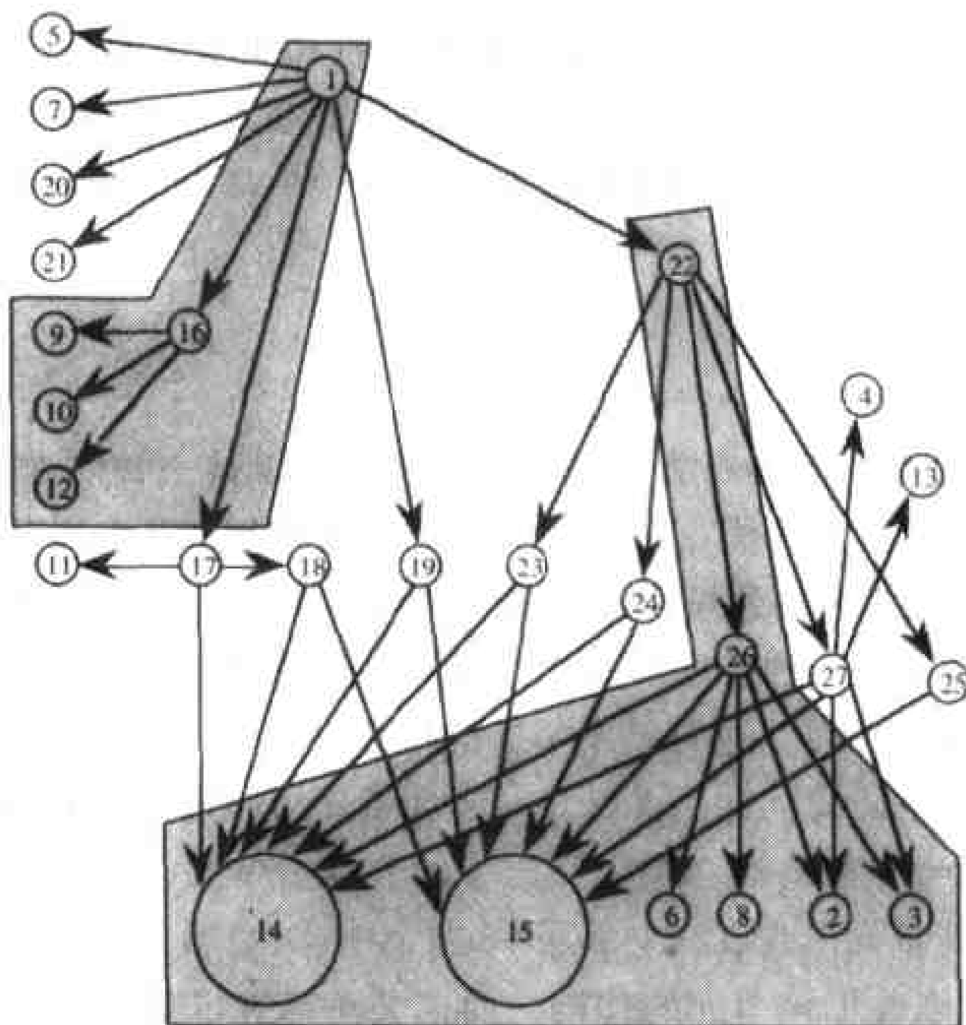


图13-7 相邻集成

表13-3 SATM邻居

节点	前驱	后继
16	1	9、10、12
17	1	11、14、18
18	17	14、15
19	1	14、15
23	22	14、15
24	22	14、15
26	22	14、15、6、8、2、3
27	22	14、15、2、3、4、13
25	22	15
22	1	23、24、26、27、25
1	---	5、7、2、21、16、17、19、22

对于给定调用图，总是可以计算出邻居数量。每个内部节点有一个邻居，如果叶节点直接连接到根节点，则还要加上一个邻居。（内部节点具有非零内度和非零外度。）我们有：

内部节点 = 节点 －（源节点 + 汇节点）

邻居 = 内部节点 + 源节点

经过合并，得到：

邻居 = 节点 － 汇节点

相邻集成可大大降低集成测试会话数量（从40降至11），并且避免了桩和驱动器的开发。最终结果是，邻居本质上是前面介绍过的三明治。（稍有不同，因为邻居的基本信息是调用图，不是分解树。）与三明治集成的共同之处更重要：相邻集成测试具有"中爆炸"集成的缺陷隔离困难。

13.3.3 优缺点

基于调用图的集成技术偏离了纯结构基础，转向行为基础，因此底层假设是一种改进。这些技术还免除了桩/驱动器开发工作量。除了这些优点之外，基于调用图的集成还与以构建和合成为特征的开发匹配得很好。例如，邻居序列可以用于定义构建。此外，我们还可以允许相邻邻居合并（合并为村庄？），并提供一种有序的基于合成的成长路径。所有这些都支持对以合成占主导地位的生命周期开发的系统，进行基于邻居的集成。

基于调用图集成测试的最大缺点，是缺陷隔离问题，尤其是对有大量邻居的情况。还有一个更微妙但是密切相关的问题。如果（当）在出现在多个邻居中的节点（单元）中发现缺陷会出现什么情况？（例如，屏幕驱动器单元出现在11个邻居中的7个中。）显然，我们要清除这个缺陷，但是这意味着以某种方式修改了该单元的代码，而这又意味着以前测试过的包含已变更代码的邻居，都需要重新进行测试。

最后，基础不确定性存在于任何测试的结构性形式：根据结构信息通过单元测试的假设，

将表现出正确的行为。我们知道测试方向：我们要让行为的系统级线索正确。当基于调用图信息的集成测试完成后，得到系统级线索还有一定距离。通过将基础从调用图信息转移到路径的特殊形式，可以解决这个问题。

13.4 基于路径的集成

数学上的进展，很大程度上得益于精细的模式：清楚地知道向哪个方向发展，然后定义向那个方向发展的概念。我们现在就为基于路径的集成测试做这样的工作。不过首先还是给出定义。

我们已经知道，单元级测试非常需要结构性测试和功能性测试的结合，对于集成（以及系统）测试来说，最好也有类似的能力。我们还知道需要用行为线索表示系统测试。最近，我们修订了集成测试的目标：不是测试单独开发并通过测试的单元之间的测试接口，而是将注意力集中在这些单元的交互上。（“协同功能”可能是确切的术语。）接口是结构性的，而交互是行为性的。

当单元执行时，要遍历一些源语句路径。假设调用沿这种路径进入另一个单元：控制经过调用的单元，到达被调用的单元，其中一些源语句的其他路径被遍历。第三部分明智地忽略了这种情况，因为现在才是讨论这个问题的更合适的时候。有两种可能：放弃单入口，单窗口要求，把这种调用看做是一个退出后接一个进入；或者抑制调用语句，因为控制最终总是要返回调用单元。抑制选择对于单元测试很有作用，但是对于集成测试正好相反。

13.4.1 新概念与扩展概念

为了达到我们的目标，需要修订部分程序图概念。与前面一样，这些程序图是指以命令式语言编写的程序。允许语句片段作为完整语句处理，语句片段是程序图中的节点。

定义

程序中的源节点是程序执行开始或重新开始处的语句片段。

单元中的第一个可执行语句显然是源节点。源节点还会出现在紧接转移控制到其他单元的节点之后。

定义

汇节点是程序执行结束处的语句片段。

程序中的最后一个可执行语句显然是汇节点，转移控制到其他单元的节点也是汇节点。

定义

模块执行路径是以源节点开始、以汇节点结束的一系列语句，中间没有插入汇节点。

目前为止的定义的作用，是现在的程序图有多个源节点和汇节点。这会大大增加单元测试的复杂性，但是集成测试假设单元测试已经完成。

定义

消息是一种程序设计语言机制，通过这种机制一个单元将控制转移给另一个单元。

取决于程序设计语言，消息可以被解释为子例程调用、过程调用和函数引用。我们约定接收消息的单元（消息的目的地）总是最终将控制返回给消息源。消息可以向其他单元传递数据。我们可以最终定义基于路径的集成测试。我们的目标是将集成测试与DD-路径做类比。

定义

MM-路径是穿插出现模块执行路径和消息的序列。

MM-路径的基本思想是，我们现在可以描述包含在单独单元之间控制转移的模块执行路径序列。这种转移是通过消息完成的，因此MM-路径永远不是可行执行路径，并且这些路径要跨越单元边界。在经过扩展的程序图中可以发现MM-路径，其中的节点表示模块执行路径，边表示消息。图13-8给出的假想例子给出了一个MM-路径（用粗线表示），表示模块A调用模块B、模块B又调用模块C。请注意，对于传统（过程）软件，MM-路径永远从主程序中开始，在主程序中结束。

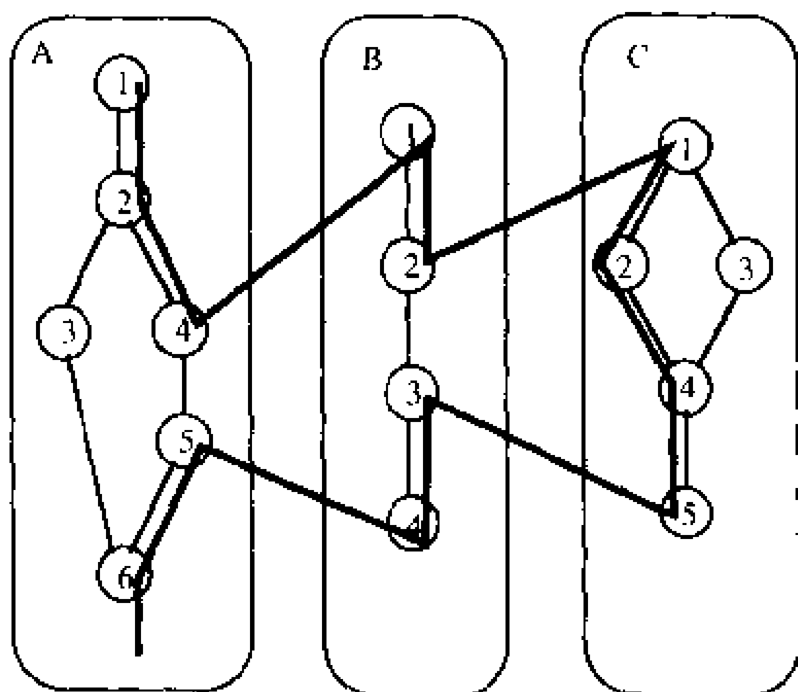


图13-8 跨三个单元的MM-路径

在模块A中，节点1和5是源节点、节点4和6是汇节点。类似地，在模块B中，节点1和3是源节点、节点2和4是汇节点。模块C只有一个源节点1和一个汇节点4。在图13-8中给出了七条模块执行路径：

MEP（A，1）=<1，2，3，6>

MEP（A，2）=<1，2，4>

MEP（A，3）=<5，6>

MEP（B，1）=<1，2>

MEP（B，2）=<3，4>

MEP（C，1）=<1，2，4，5>

MEP（C，2）=<1，3，4，5>

下面可以定义一种集成测试，这种集成测试能够与非常有效地用于单元测试的DD-路径做类比。

定义

给定一组单元，其MM-路径图是一种有向图，其中的节点表示模块执行路径，边表示消息和单元之间的返回。

请注意，MM-路径图是按照一组单元定义的。这直接支持单元合成和基于合成的集成测试。我们甚至可以合成到下层单个模块执行路径，不过可能太详细，没有必要。

图13-9给出了图13-8所示例子的MM-路径图。实线箭头表示消息，相应的返回由虚线箭头表示。我们应该考虑模块执行路径、程序路径、DD-路径和MM-路径之间的关系。程序图是DD-路径序列，MM-路径是模块执行路径序列。但是，DD-路径和模块执行路径之间没有简单的对应关系，两者可能相互包容，更有可能部分重叠。由于MM-路径实现超出单元边界的功能，因此确实有一种关系：MM-路径与单元的交叉。这种交叉中的模块执行路径，可以与（MM-路径）函数片类比。换句话说，这种交叉中的模块执行路径，是模块执行路径所在单元的功能约束。

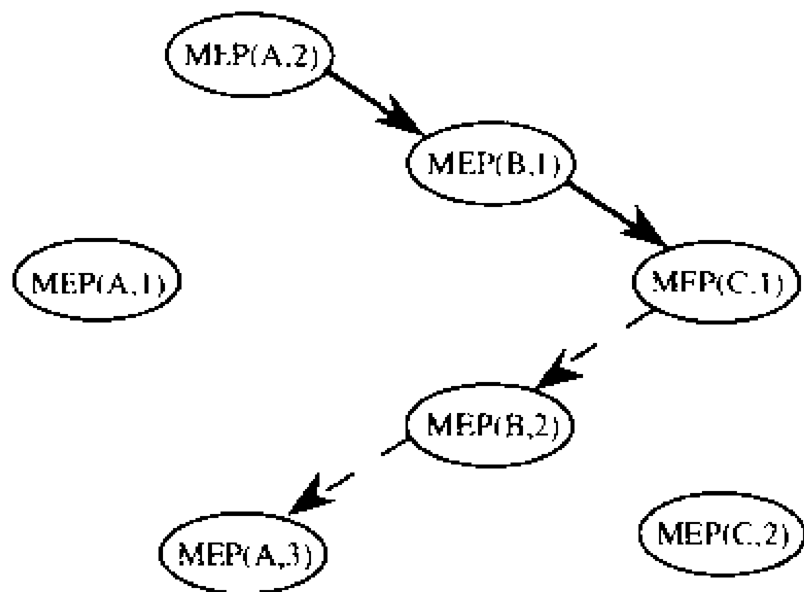


图13-9　从图13-8中导出的MM-路径图

MM-路径定义需要具有某种实际指导方针。MM-路径有多长（“深”可能更确切）？MM-路径末端点有两点可观察的行为准则：消息和数据静止。当到达不发送消息的节点时，消息静止发生（例如图13-8中的模块C）。

当处理不立即使用的存储数据的创建的序列结束时，数据静止发生。在ValidateCard单元中，获取账户余额，但是直到PIN输入成功后才使用。图13-10给出了数据静止如何出现在传统数据流图中。静止点是MM-路径的自然端点。

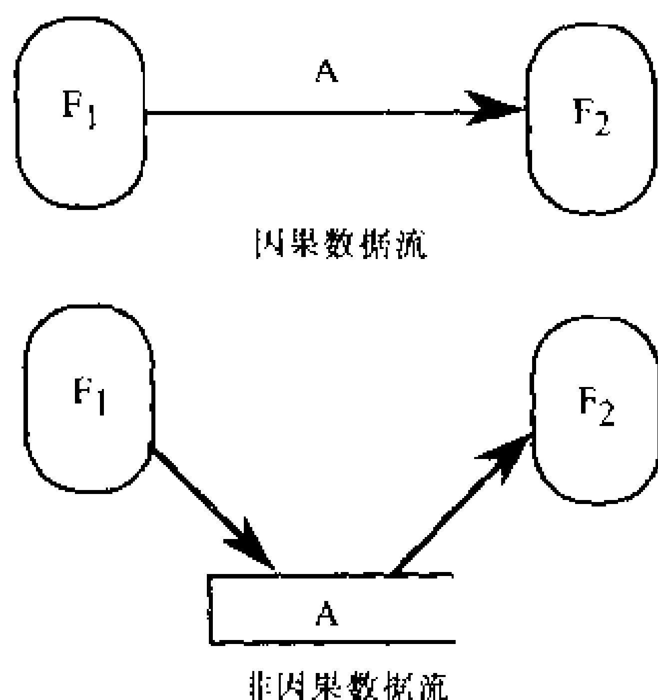


图13-10　数据静止

13.4.2　SATM系统中的MM-路径

为了方便起见，这里再重复一次第12章开发的伪代码描述，语句片段的编号方法与构造程序图中的编号方法相同。此外，消息编号在注释中做了说明。我们将使用这些编号描述所选的MM-路径。ScreenDriver参数引用图12-6中带编号的屏幕。过程GetPIN实际上是桩，用于响应ExpectedPIN = 1234的正确数字事件序列。

```
1   Main Program
2.  State = AwaitCard
3   Case State
4.  Case 1: AwaitCard
5.          ScreenDriver(1, null)                  msg 1
6.          WatchCardSlot (CardSlotStatus)         msg 2
7.          Do While CardSlotStatus is Idle
8.              WatchCardSlot (CardSlotStatus)     msg 3
9.          End While
10.         ControlCardRoller(accept)              msg 4
11.         ValidateCard(CardOK, PAN)              msg 5
12.         If CardOK
13.             Then State = AwaitPIN
14.             Else ControlCardRoller(eject)      msg 6
15.         EndIf
16.         State = AwaitCard
17. Case 2: AwaitPIN
18.         ValidatePIN(PINok, PAN)                msg 7

19.         If PINok
20.             Then    ScreenDriver(2, null)      msg 8
21.                     State = AwaitTrans
22.             Else    ScreenDriver(4, null)      msg 9
23.         EndIf
24.         State = AwaitCard
25. Case 3: AwaitTrans
```

```
            ManageTransaction                                msg 10
            State = CloseSession
Case 4: CloseSession
            If NewTransactionRequest
                Then    State = AwaitTrans
                Else    PrintReceipt                         msg 11
            EndIf
            PostTransactionLocal                             msg 12
            CloseSession                                     msg 13
            ControlCardRoller(eject)                         msg 14
            State = AwaitCard
End Case (State)
End     (Main program SATM)

Procedure ValidatePIN(PINok, PAN)
GetPINforPAN(PAN, ExpectedPIN)                               msg 15
Try = First
Case Try of
Case 1: First
        ScreenDriver(2, null)                                msg 16
        GetPIN(EnteredPIN)                                   msg 17
        If EnteredPIN = ExpectedPIN
            Then    PINok = True
            Else    ScreenDriver(3, null)                    msg 18
                    Try = Second
        EndIf
Case 2: Second
        ScreenDriver(2, null)                                msg 19
        GetPIN(EnteredPIN)                                   msg 20
        If EnteredPIN = ExpectedPIN
            Then    PINok = True
            Else    ScreenDriver(3, null)                    msg 21
        EndIf
        Try = Third
Case 3: Third
        ScreenDriver(2, null)                                msg 22
        GetPIN(EnteredPIN)                                   msg 23
        If EnteredPIN = ExpectedPIN
            Then    PINok = True
            Else    ScreenDriver(4, null)                    msg 24
                    PINok = False
        EndIf
EndCase (Try)
End.        (Procedure ValidatePIN)

Procedure GetPIN(EnteredPIN, CancelHit)
Local Data: DigitKeys = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
CancelHit = False
EnteredPIN = null string
DigitsRcvd=0
Do While NOT(DigitsRcvd=4 OR CancelHit)
    KeySensor(KeyHit)                                        msg 25
    If KeyHit IN DigitKeys
        Then
            EnteredPIN = EnteredPIN + KeyHit
            INCREMENT(DigitsRcvd)
```

```
80.             If digitsRcvd = 1
81.                 Then ScreenDirver (2, 'X---')       msg 26
82              EndIf
83.             If digitsRcvd = 2
84.                 Then ScreenDirver (2, 'XX--')       msg 27
85.             EndIf
86.             If digitsRcvd = 3
87.                 Then ScreenDirver (2, 'XXX-')       msg 28
88              EndIf
89              If digitsRcvd = 4
90.                 Then ScreenDirver (2, 'XXXX')       msg 29
91.             EndIf
92.         Else
93.             CancelHit = True
94.     EndIf
95. End While
96. End.  (Procedure GetPIN)
```

SATM 主程序包含16个源节点。除节点1的所有节点都是过程/函数调用返回控制的地点：1、6、7、9、11、12、15、19、21、23、27、30、32、34、35和36。SATM 主程序包含16个汇节点。对于源节点，大多数节点都是过程/函数调用点：5、6、8、10、11、14、17、18、20、22、26、31、33、34、35和38。请注意，如果使用两个串行过程调用，则语句可以既是汇节点，又是源节点。SATM主程序中的大多数模块执行路径都很短，出现这种情况是因为到其他单元的消息密度很高。

在SATM 主程序头17条线中，只包含一条非平凡模块执行路径：<1，2，3，4>。过程调用，例如<5>、<6>、<8>、<10>、<11>和<14>，都是平凡的，因为它们在SATM 主程序中发生不多。其他很短的模块执行路径与控制结构关联，例如<9，7>、<9，10>和<15，16>。

以下是第一次尝试正确PIN输入的MM-路径。模块执行路径由单元名后接语句片段编号序列描述。图13-11说明了使用UML风格序列图的MM-路径串行性质。

```
Main (1, 2, 3, 17, 18)
msg 7
ValidatePIN (39, 40)
msg 15
GetPINforPAN (no pseudo-code given)
ValidatePIN (41, 42, 43, 44)
msg 16
ScreenDriver (no pseudo-code given)
ValidatePIN (45)
msg 17
GetPIN(69, 70, 71, 72, 73, 74, 75)
msg 25
KeySensor (no pseudo-code given)      'first digit
GetPIN (76, 77, 78, 79, 80, 81)
 msg 26
ScreenDriver (no pseudo-code given)
GetPIN (82, 83, 85, 86, 88, 89, 91, 94, 95, 74, 75)
msg 25
KeySensor (no pseudo-code given) )    'second digit
GetPIN (76, 77, 78, 79, 80, 82, 83, 84)
```

msg 27
ScreenDriver (no pseudo-code given)
GetPIN (85, 86, 88, 89, 91, 94, 95, 74, 75)
msg 25
KeySensor (no pseudo-code given)) 'third digit
GetPIN (76, 77, 78, 79, 80, 82, 83, 85, 86, 87)
msg 28
ScreenDriver (no pseudo-code given)
GetPIN (88, 89, 91, 94, 95, 74, 75)
msg 25
KeySensor (no pseudo-code given)) 'fourth digit
GetPIN (76, 77, 78, 79, 80, 82, 83, 85, 86, 88, 89, 90)
msg 29
ScreenDriver (no pseudo-code given)
GetPIN (91, 94, 95, 74, 96)
ValidatePIN (46, 47, 50, 67, 68)
Main(19)

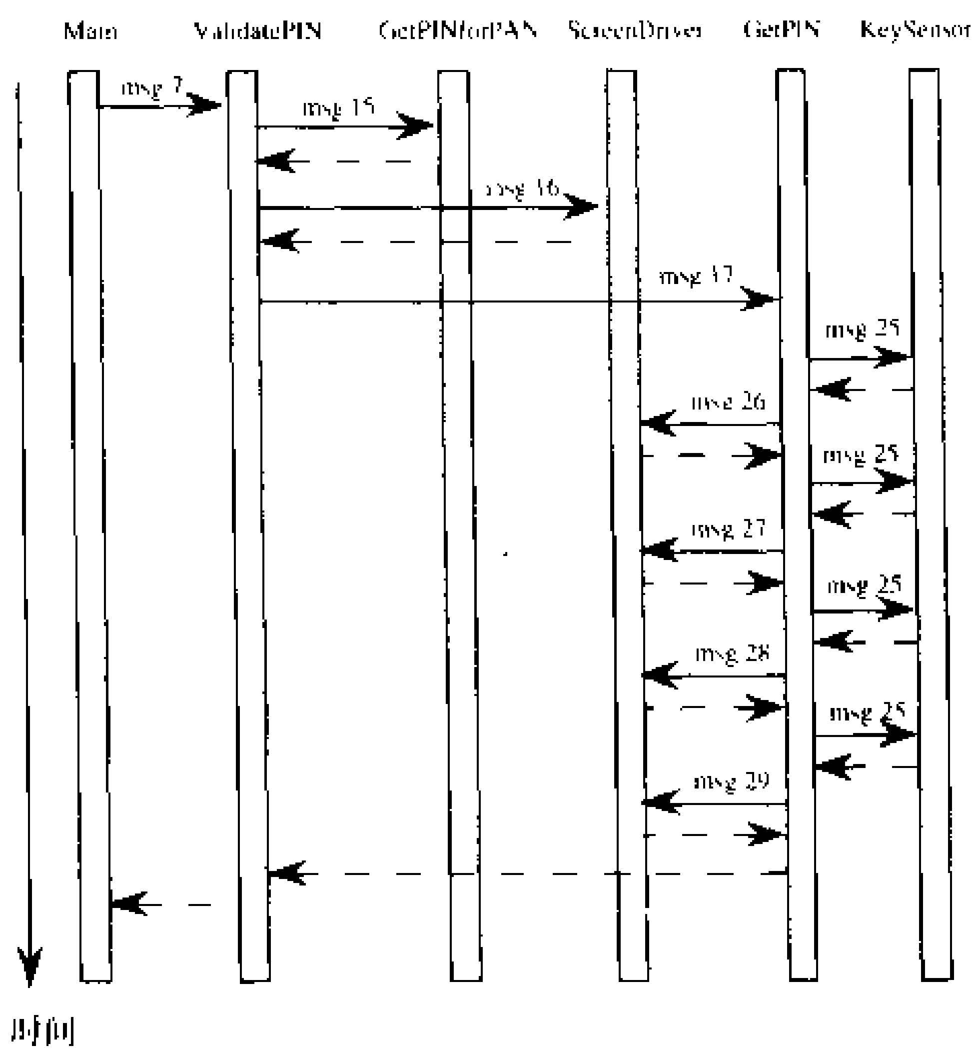


图13-11 示例MM-路径的UML序列图

13.4.3 MM-路径复杂度

如果比较图13-8和图13-11中的MM-路径，就会直观地发现后者要比前者复杂得多。这两张图的有向图一起显示在图13-12中。多条边（例如ScreenDriver和GetPIN之间的多条边）

表示消息连接，双箭头表示消息的发送和返回（比较清晰）。由于是强连接有向图，因此可以“探索地”计算其圈复杂度。前面提到过，圈复杂度的计算公式为：

$$V(G) = e - n + 2p$$

其中，p是强连接区域的个数。对于结构化过程代码，永远有p = 1，因此这个公式简化为V（G）= e－n＋2。根据这个公式可得到图13-12的圈复杂度分别为V（G）= 3和V（G）= 20

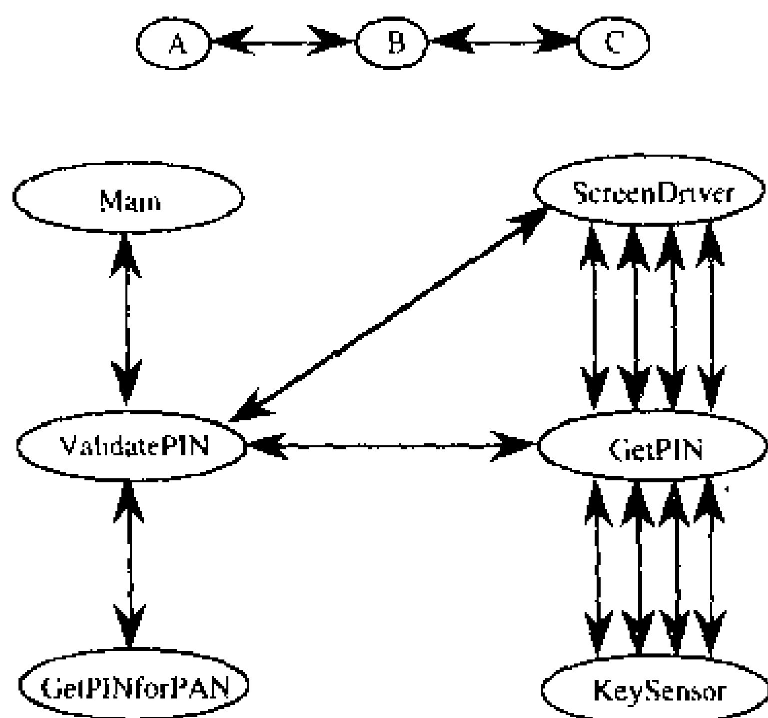


图13-12 MM-路径有向图

这看起来是合理的。第二张图显然比第一张图更复杂，如果删除部分消息，比方说GetPIN和KeySensor节点之间的消息，复杂度在直观上和计算上都被降低。公式中“2p”的角色比较讨厌，它的作用是修正公式，但是总等于2，所以直接删除p。如果删除p，则单元A调用单元B，并且B返回这个最简单的MM-路径，复杂度为0。更糟糕的是，独立单元会有负的复杂度，即-1。通过本章练习还会得到其他复杂度。

13.4.4 优缺点

MM-路径是功能性测试和结构性测试的一种混合。在表达输入和输出行动上，MM-路径是功能性的，因此所有功能性测试技术都是潜在可使用的。在标识方式上，特别是MM-路径图的标识方式上，它是结构性的。总效果是，功能和结构方法的交叉检查被结合到基于路径集成测试的构造中。因此，我们可以避免结构性测试的缺点，同时又使集成测试与系统测试无缝连接。基于路径的集成测试既适用于采用传统瀑布过程开发的软件，也同样适用于采用某种基于合成可选的生命周期模型开发的软件。第18章还要讨论这个问题，那时将会看到基于路径的集成测试同样也适用于面向对象的软件测试。基于路径集成测试最重要的优点，在于它与实际系统行为密切匹配，而不是靠基于分解和调用图集成的结构性推动。

基于路径集成测试的优点也是有代价的：需要更多的工作量标识MM-路径。这种工作量可能会与桩的消除和驱动器开发所需工作量有偏差。

13.5 案例研究

这里重新编写读者已经熟悉了的Next Date例子，将主程序的功能分解为过程和函数。这个"集成版本"有一点扩展：对月份、日期和年增加了（有限的）有效性检查，伪代码从50行语句增长到81行。图13-13和图13-14给出了NextDate集成版本的单元程序图，功能分解在图13-15中表示，图13-16给出的是"调用图"。

```
1   Main integrationNextDate
        Type   Date
                  Month As Integer
                  Day As Integer
                  Year As Integer
        EndType
        Dim today As Date
        Dim tomorrow As Date
2       GetDate(today)                                   'msg1
3.      PrintDate(today)                                 'msg2
4       tomorrow = IncrementDate(today)                  'msg3
5.      PrintDate(tomorrow)                              'msg4
6   End Main

7   Function isLeap(year)  Boolean
8.      If (year divisible by 4)
9.          Then
10.             If (year is NOT divisible by 100)
11.                 Then isLeap   True
12.                 Else
13.                     If (year is divisible by 400)
14.                         Then isLeap = True
15                          Else isLeap = False
16.                     EndIf
17.             EndIf
18.         Else isLeap   False
19.     EndIf
20. End (Function isLeap)

21. Function lastDayOfMonth(month, year)     Integer
22      Case month Of
23.         Case 1  1, 3, 5, 7, 8, 10, 12
24.             lastDayOfMonth = 31
25.         Case 2: 4, 6, 9, 11
26.             lastDayOfMonth = 30
27.         Case 3. 2
28.             If (isLeap(year))                        'msg5
29                  Then lastDayOfMonth = 29
```

```
                Else lastDayOfMonth = 28
            EndIf
    EndCase
End (Function lastDayOfMonth)

Function GetDate(aDate)        Date
    dim aDate As Date
    Function ValidDate(aDate) Boolean     'within scope of GetDate
        dim aDate As Date
        dim dayOK, monthOK, yearOK As Boolean

        If ((aDate.Month > 0) AND (aDate.Month <=12)
            Then monthOK = True
            Else monthOK = False
        EndIf

        If (monthOK)
            Then
                If ((aDate.Day > 0) AND
                        (aDate.Day <= lastDayOfMonth(aDate.Month, aDate.Year))'msg6
                    Then dayOK = True
                    Else dayOK = False
                EndIf
            EndIf

    If ((aDate.Year >= 1811) AND (aDate.Year <= 2012)
            Then yearOK = True
            Else yearOK = False
            EndIf

    If (monthOK AND dayOK AND yearOK)
            Then ValidDate = True
            Else ValidDate = False
        EndIf
        End (Function ValidDate)
  ' GetDate body begins here
    Do
        Output("enter a month")
        Input(aDate.Month)
        Output("enter a day")
        Input(aDate.Day)
        Output("enter a year")
        Input(aDate.Year)
        GetDate.Month = aDate.Month
        GetDate.Day = aDate.Day
        GetDate.Year = aDate.Year
    Until (ValidDate(aDate))                         'msg7
```

```
67.     End (Function GetDate)

68. Function IncrementDate(aDate)    Date
69.     If (aDate.Day < lastDayOfMonth(aDate.Month))    'msg8
70.         Then aDate.Day = aDate.Day + 1
71.         Else aDate.Day = 1
72.             If (aDate.Month = 12)
73.                 Then aDate.Month = 1
74.                     aDate.Year = aDate.Year + 1
75.                 Else aDate.Month = aDate.Month + 1
76.             EndIf
77.     EndIf
78. End (IncrementDate)

79. Procedure PrintDate(aDate)
80.     Output( "Day is ", aDate.Month, "/", aDate.Day, "/", aDate.Year)
81. End (PrintDate)
```

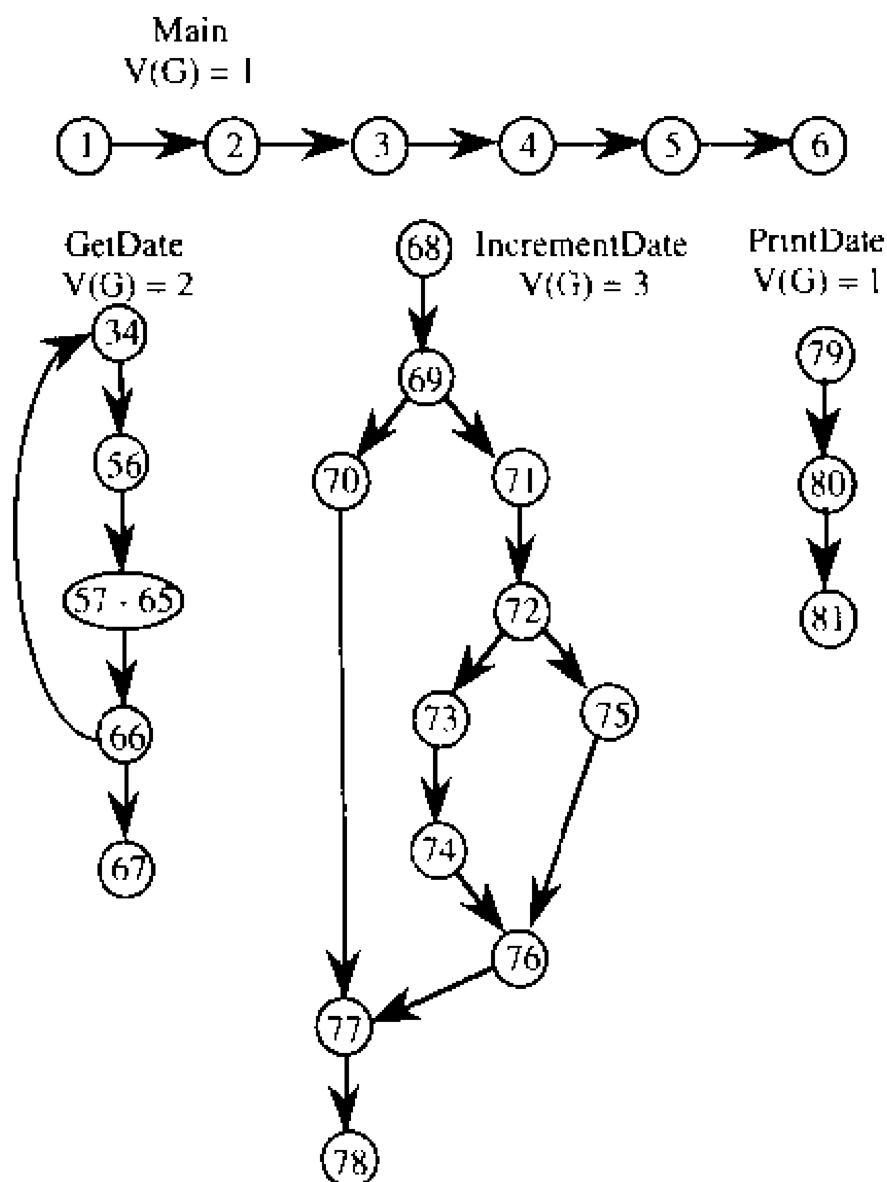


图13-13 主程序和第一层单元

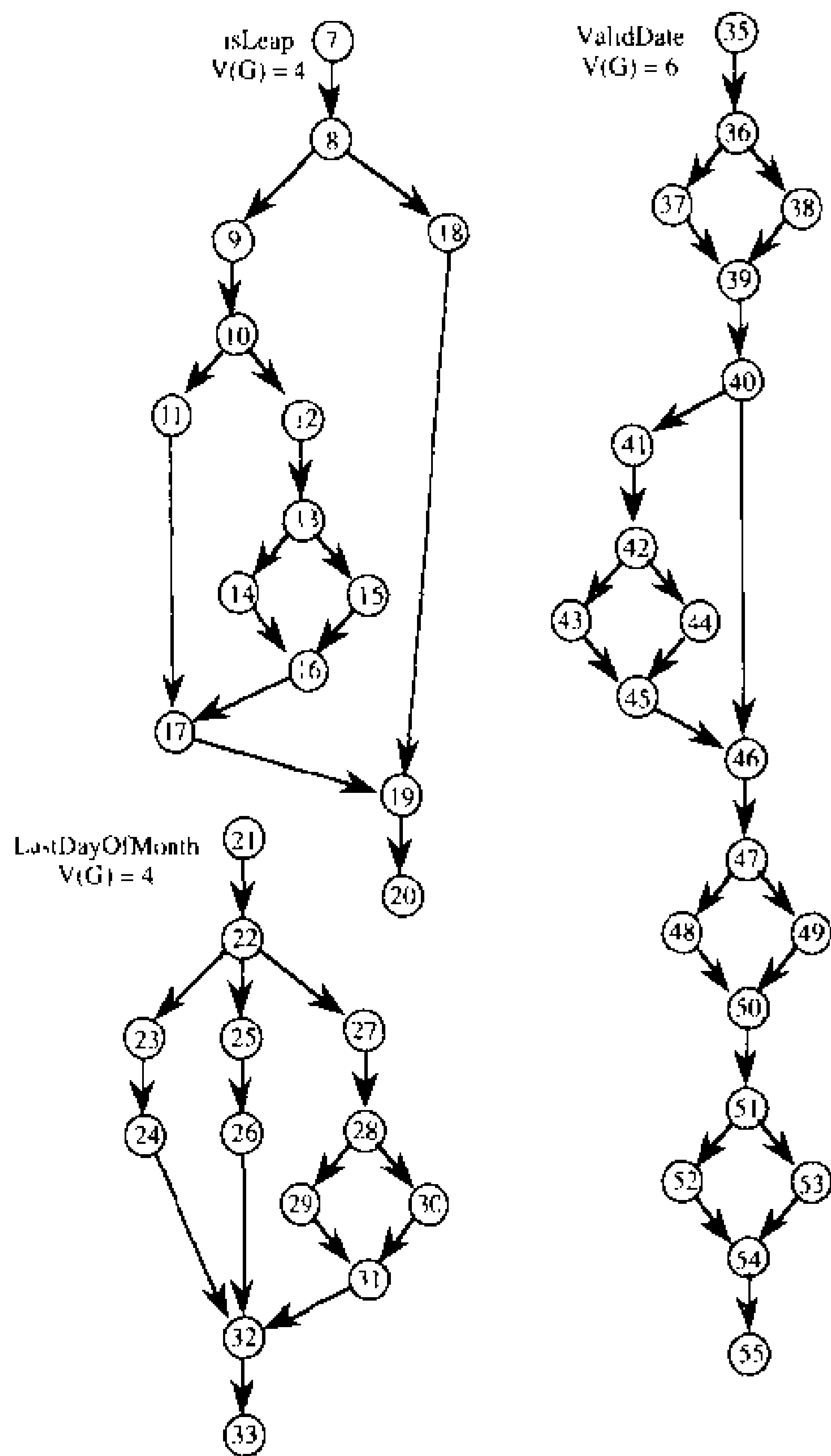


图13-14　低层单元

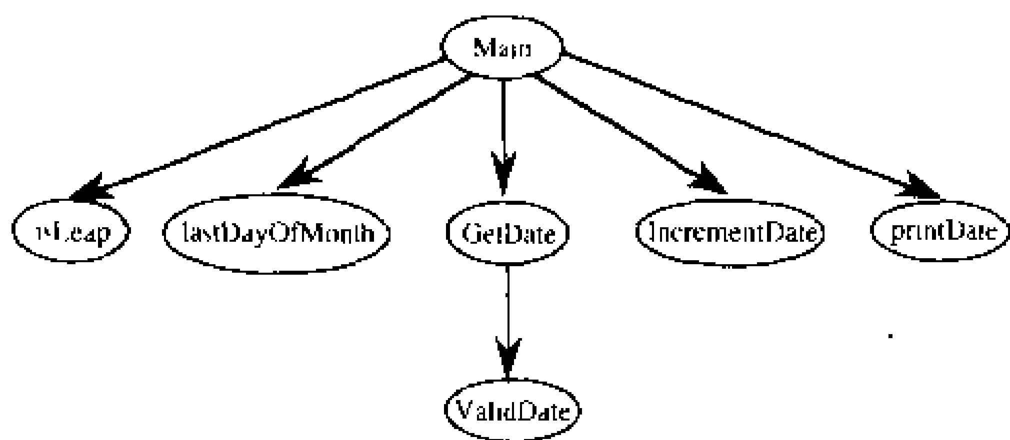


图13-15　集成版本的功能分解

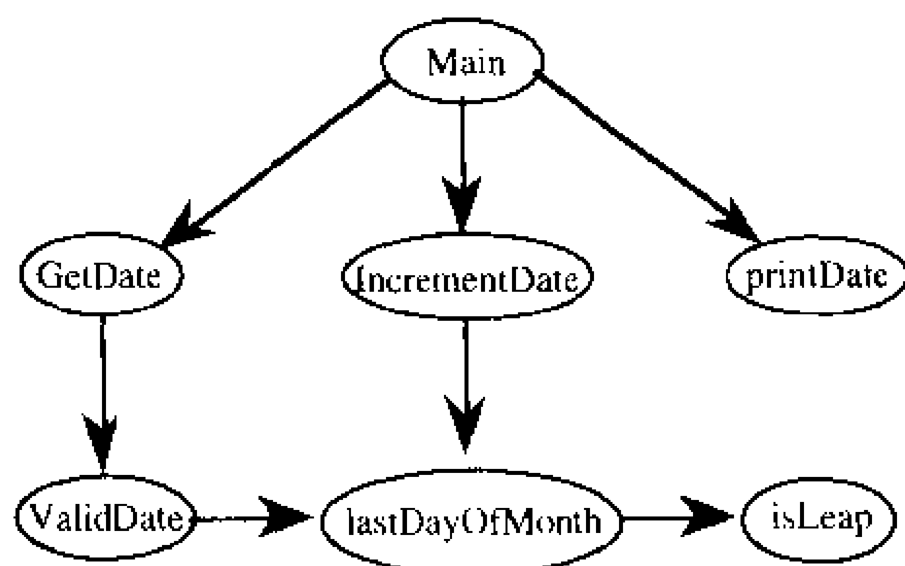


图13-16 集成版本的调用图

13.5.1 基于分解的集成

函数isLeap（是闰年）和lastDayOfMonth（月份最后日期）位于分解的第一层，因为必须向GetDate（取日期）和IncrementDate（日期增1天）提供服务（可以把isLeap包含在lastDayOfMonth内部）。图13-15所示的基于分解的成对集成是有问题的，函数isLeap和lastDayOfMonth从来都没有被主程序直接调用，因此这些集成过程是空的。自底向上成对集成从isLeap开始，然后是lastDayOfMonth。ValidDate（检验日期）和GetDate将会被用到。包括主程序和GetDate、IncrementDate和PrintDate（打印日期）的函数对都是有用的（但是很短）会话。构建ValidDate和lastDayOfMonth的桩会很容易。

13.5.2 基于调用图的集成

如图13-16所示的基于调用图的成对集成，是对基于分解的成对集成的改进。很明显没有了空的集成会话，因为边引用的是实际单元。桩仍然有问题。三明治集成是合适的，因为这个例子很小。事实上，三明治集成自己就可以产生构建序列。构建1可以包含主程序和PrintDate，构建2可以包含主程序、IncrementDate、lastDayOfMonth，并且IncrementDate已经提供给PrintDate。最后，构建3增加其余的单元，即GetDate和ValidDate。

基于"调用图"的相邻集成可以通过ValidDate和lastDayOfMonth的邻居进行，接下来可以集成GetDate和IncrementDate的邻居，最后可以集成主程序的邻居。请注意，这些邻居构成一种构建序列。

13.5.3 基于MM-路径的集成

由于程序是数据驱动的，因此所有MM-路径都要从主程序开始，并回到主程序。以下是2002年5月27日的第一条MM-路径（当主程序调用PrintDate和IncrementDate时，还有其他MM-路径）。

```
Main (1, 2)
    msg1
    GetDate (34, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66)
        msg7
        validDate (35, 36, 37, 39, 40, 41, 42)
            msg6
            lastDayOfMonth (21, 22, 23, 24, 32, 33)'point of message quiescence
        ValidDate (43, 45, 46, 47, 48, 50, 51, 52, 54, 55)
    GetDate (67
Main (3)
```

请注意，语句片段序列（如图13-13和13-14所示）标识出从源节点到汇节点的所有路径。在消息静止点上这是直接的。在其他单元中，节点序列对必须级联起来，构成完整的从源节点到汇节点的路径。现在，在描述有多少条MM-路径已经足够这个问题上，我们处于有利地位：MM-路径集合应该覆盖单元集合中所有从源到汇节点的路径。如果存在循环，则要进行压缩，产生有向无环路图，因此可解决潜在无限（或非常多）条路径问题。

13.6 参考文献

Fordahl, Matthew, Elementary Mistake Doomed Mars Probe, The Associated Press, Oct. 1, 1999; also, www.fas.org/mars/991001/~mars01.htm

13.7 练习

1. 请找出ValidatePIN和GetPIN中的源节点和汇节点。
2. 请找出ValidatePIN的模块执行路径。
3. 以下是MM-路径的一些其他可能的复杂度指标：

V（G）= e – n

V（G）= 0.5e – n + 2

节点外度之和

节点数加边数

请构建一些例子，并计算这些指标的复杂度。观察是否能够得到说明性的值。

第14章

系统测试

在测试的三级中，系统级测试是最接近日常测试实践的。我们测试很多东西：在购买二手车时要进行系统测试，在订购在线网络服务时要进行系统测试，等等。这些大家所熟悉的测试形式的共同模式，是根据我们的预期评估产品，而不是根据规格说明或标准。结果，测试目标不是找出缺陷，而是证明其性能。因此，系统测试是从功能出发，而不是从结构出发。由于大家对系统测试有很直观的认识，因此系统测试在实践中不如本来应该的那么正式，常常要在产品交付截至日期之前压缩系统测试时间。

我们继续使用工艺师这种类比。第12章已经提到过，我们需要更好地理解媒介，要以系统级行为的线索观察系统测试。本章首先介绍一种新的构造，即原子系统功能，然后进一步讨论线索概念，强调基于线索系统测试的一些实践问题。系统测试与需求规格说明密切相关，因此本章将讨论如何以通用表示法找出线索。所有这些对于基于线索的系统测试策略都是必要的，这种策略要探索功能性测试和结构性测试的协作。我们最后将这种策略用于简单自动柜员机（SATM）系统例子中。

14.1 线索

线索（thread）很难定义。事实上，一些已经公开的定义都是矛盾、容易产生误导或错误的。可以把线索看做是一种不需要形式化定义的原始概念。本章将使用例子开发一个"共享构想"。以下是对线索的多种看法：

- 一般使用的场景。
- 系统级测试用例。
- 激励/响应对。
- 由系统级输入序列产生的行为。
- 端口输入和输出事件的交替序列。
- 系统状态机描述中的转换序列。
- 对象消息和方法执行的交替序列。
- 机器指令序列。
- 源指令序列。
- MM-路径序列。
- 原子系统功能序列。

线索有不同层次。单元级线索通常被理解为源指令执行时间路径，或DD-路径。集成级线索是MM-路径，即模块执行和消息交替序列。如果延续这种模式，则系统级线索就是原子系统功能（稍后定义）序列。由于原子系统功能在输入和输出时有端口事件，因此原子系统功能序列意味着端口输入和输出事件的交替序列。最终结果是，线索提供三层测试的统一视图。单元测试进行单个函数测试，单元之间集成测试检查交互，系统测试检查原子系统功能之间的交互。本章重点介绍系统级线索，并回答一些基本问题，例如"线索有多大？到哪里寻找线索？怎样测试线索？"

14.1.1 线索的可能性

定义系统级线索的端点有点麻烦。我们通过使用线索要达到的目标，反向给出一种清晰的基于图论的定义。以下是SATM系统的四个候选线索：

- 数字输入。
- 个人标识编号输入（PIN）
- 简单事务："ATM卡输入"，"PIN输入"，选择事务处理类型（存款、取款）、提供账户细节（支票账户或储蓄账户、金额）、引导操作、报告结果
- 包含两个或多个简单事务处理的ATM会话。

数字输入是最小原子系统功能的一个好例子。它以一个端口输入事件开始（键入数字），以一个端口输出事件结束（屏幕显示数字），因此是激励/响应对。如果回到第13章中的例子，就会看到这种原子系统功能（ASF）就是详细列出的MM-路径示例的子路径。这种粒度对于系统测试太细，其粒度适合集成测试。

第二个候选线索"PIN输入"是集成测试上限，同时又是系统测试起点的一个好例子。"PIN输入"是原子系统功能的一个好例子，也是激励/响应对系列的一个好例子（由端口输入事件发起的系统级行为，遍历一些编程实现的逻辑，以多种可能应答中的一个结束（端口输出事件））。第13章已经介绍过，"PIN 输入"需要一系列系统级输入和输出。

1. 显示请求输入PIN数字的屏幕。
2. 数字输入和屏幕显示的交替序列。
3. 客户在输入完完整的PIN之前进行取消的可能性。
4. 系统处理：客户有三次机会输入正确的PIN。一旦输入正确的PIN，用户会看到请求输入事务处理类型的屏幕；否则显示提示客户ATM卡将不被返回的屏幕，不提供ATM功能。

这显然属于系统级测试领域，几个激励/响应对也很明显。原子系统功能的其他例子包括"ATM卡输入"、"事务处理选择"、"提供事务细节"、"事务处理报告"和"会话结束"。这些原子系统功能是最大的集成测试单元，最小的系统测试单元。也就是说，不应该集成

大于原子系统功能的单元，同时也不对小于原子系统功能的单元进行系统测试

第三个候选线索简单事务处理，具有“端到端”的完整性。客户永远不能单独执行PIN输入（需要Card Entry），但是这个简单事务处理常常执行。这是系统级线索的一个好例子，请注意，它涉及多个原子系统功能的交互。

最后一个候选线索（会话）实际上是一系列线索。这也是系统测试的一个合适部分，在这个层次上，我们对线索之间的交互感兴趣。但是，大多数系统测试成果从来也不能达到线索交互层次。（更多内容，请参阅第15章。）

14.1.2 线索定义

本节通过定义有助于达到期望目标的新术语来简化讨论。

定义

原子系统功能（ASF）是一种在系统层可以观察得到的端口输入和输出事件的行动

在事件驱动系统中，ASF由事件静止点分开，当系统（接近）空闲、等待端口输入事件以触发进一步处理时，会出现这种情况。事件静止有一种有意思的Petri网解释 在传统Petri网中，当无法转换时会出现死锁。在事件驱动Petri网中，事件静止类似于死锁，但是输入事件可以为Petri网注入活力。SATM系统多处有事件静止：一处是SATM主程序开始时的紧循环，系统显示欢迎屏幕，并等待在ATM卡槽中输入卡 事件静止是系统级特性，可与集成级的消息静止类比。

ASF的事件静止具有与MM-路径的消息静止一样的作用：它提供一种自然端点. ASF开始于一个端口输入事件，遍历一个或多个MM-路径的一部分，以一个端口输出事件结束 当在系统层上观察时，并没有特别理由将ASF向下分解（因此叫做原子），在SATM系统中，数字输入是ASF的一个好例子，同样，ATM输入、现金给付和会话关闭也都是ASF的好例子 “PIN 输入”可能太大，也许应该叫做分子系统功能

原子系统功能表示集成测试与系统测试之间的缝隙，是集成测试的最大测试项，是系统测试的最小测试项。可以在两个级别上测试ASF。同样，数字输入ASF是一个好例子（请参阅第13章中的MM-路径例子）。在系统测试过程中，端口输入事件是由KeySensor检测到的按下物理键，并作为字符串变量发送给GetPIN。（请注意，KeySensor执行物理到逻辑的转换。）GetPIN确定是否按下数字键或取消键，并做出相应的应答。（请注意，忽略按下按钮动作。）ASF最后显示屏幕2或4，不是要求系统输入键并在屏幕上显示，而是使用驱动器提供这些功能，并通过集成测试检查数字输入ASF.

有意思的ASF包含在ValidatePIN中，这个单元控制与PIN输入过程有关的所有屏幕显示 它首先显示屏幕2（要求客户输入PIN）。然后调用GetPIN，系统事件静止，直到客户击键 客户击键会启动我们刚刚讨论过的GetDigit ASF。这里我们发现了一个奇怪的集成缺陷。请注意，屏幕2在两个地方显示：一是GetPIN while循环中的Then子句，二是ValidatePIN每个Case子句的第一个语句。通过从GetPIN中删除屏幕显示，并直接返回要显示的字符串（例如

"X—"）可以解决这个问题。

参考第13章伪代码例子，可发现在语句75~93中包含了四个ASF：每个ASF都以观察端口输入事件（客户击键）的KeySensor开始，以端口输出事件（有不同PIN回显的ScreenDriver调用）的密集发生结束。可以把这四个ASF命名为GetDigit1、GetDigit2、GetDigit3和GetDigit4，这些ASF略有不同，因为后面的ASF包含前面的If语句（这个模块可能要返工，使得while循环重复单一的ASF。）

SATM系统的这个部分，还说明单元测试和集成测试之间的差别。当GetPIN被单元测试时，其输入来自KeySensor（起输入语句的作用）。GetPIN的输入空间包含数字0~9和取消键（可以作为字符串或字符数据处理。）我们可以为该函数增加键B1、B2和B3。如果这样做，则传统等价类测试会是一种很好的选择。我们测试的功能是，GetDigit是否将输入的键重新构造到数字串中，以及取消键的布尔指示是否正确。

定义

给定通过原子系统功能定义的一个系统，系统的ASF图是一种有向图，其中的节点表示ASF，边表示串行流。

定义

源ASF是一种原子系统功能，在系统ASF图中作为源节点出现。类似地，汇ASF也是一种原子系统功能，在系统ASF图中作为汇节点出现。

在SATM系统中，"ATM卡输入"是源ASF，会话结束ASF是汇ASF。请注意，中间ASF永远不会在系统级本身测试，它们需要通过前驱ASF"实现其目标"。

定义

系统线索在系统的ASF图中，是一种从源ASF到汇ASF的路径。

定义

给定通过系统线索定义的一个系统，系统的线索图是一种有向图，节点表示系统线索，边表示单个线索的顺序执行。

这组定义提供了线索更广的视图的内聚集，以非常短的线索开始（在一个单元内），以系统级线索之间的交互结束。我们使用这些视图方式很像使用显微镜上的目镜，通过改变目镜，可以看到不同粒度的物体。提出这些概念还只是问题的一部分，支持这些概念则是另外一回事。以下我们以测试人员对需求规格说明的观点，研究如何标识线索。

14.2　需求规格说明的基本概念

请回忆向量空间的基本概念：向量空间是一组可以生成空间中所有元素的独立元素。本节并不想讨论大量的需求规格说明方法、标记和技术，而是根据一组基本需求规格说明构造，即数据、行动、设备、事件和线索（Jorgensen，1989），重点讨论系统测试。每个系统都可以这五种基本概念表示（而且所有需求规格说明技术都是这些概念的某种组合）。这里

讨论这些基本概念，是为了说明它们是如何支持测试人员的线索标识过程的

14.2.1 数据

当系统以其数据描述时，关注的是系统所使用和创建的信息 我们采用变量、数据结构、字段、记录、数据存储和文件来描述数据。实体/关系模型是高层数据描述的最常见的选择，在更细的层次上使用一些常规表达式（例如Jackson图或数据结构图） 以数据为核心的观点还是面向对象分析一些优点的切入点 数据是指经过初始化、存储、更新或（可能）销毁的信息 在SATM系统中，初始数据描述各种账户（PAN）及其PIN，每个账户都有一个数据结构，包含诸如账户余额这样的信息 当出现ATM事务处理时，结果作为被创建的数据保存，并在每天将终端数据报告给中央银行时使用 很多系统都是以数据为中心的观点占主导地位，这些系统常常以CRUD行动开发（“创建、“检索”、“更新”、“删除”） 我们将以这种方式描述SATM系统的事务处理部分，但是这种方式很难描述用户界面部分

有时线索可以直接通过数据模型标识 数据实体之间的关系可以是一对一、一对多、多对一或多对多，这些差别在处理数据的线索中都有应用。例如，如果银行客户可以拥有多个账户，每个账户需要惟一PIN，如果多人可以访问同一个账户，则需要具有相同PAN的ATM卡。我们还可以找到只读取但从来不写入的初始数据（例如PAN和“预期PIN”对偶），这种只读数据必须是系统初始化过程的一部分。如果不是，那么必须有创建这种数据的线索。因此，只读数据是一种源ASF指示器。

14.2.2 行动

以行动为中心建模仍然是需求规格说明的一种常见形式，这是因为有命令式程序设计语言以行动为中心性质的历史原因。行动有输入和输出，这些输入和输出既可以是数据，也可以是端口事件。以下是一些与具体方法学有关的行动同义词：转换、数据转换、控制转换、处理、活动、任务、方法和服务。行动还可以分解为低层活动，例如第12章中的数据流图。行动的输入/输出视图正好是功能性测试的基础，行动的分解（以及最终实现）则是结构性测试的基础。

14.2.3 设备

每个系统都有端口设备，这些端口设备是系统级输入和输出（端口事件）的源和目的地 区分端口和端口设备之间的微小差别有时对测试人员很有帮助 在技术上，端口是I/O设备接入系统的点，例如串行端口和并行端口、网络端口、电话端口 物理行动（击键和屏幕发光）也在端口设备上发生，要从物理行动转换为逻辑行动（或从逻辑行动转换为物理行动） 如果没有实际端口设备，系统测试可以通过“将端口边界向内移动”到端口事件的逻

辑实例上实现，从现在开始，我们将只使用术语"端口"代替端口设备．SATM系统中的端口包括数字和取消键、功能键、显示屏幕、存款和取款通道、ATM卡和收据槽，以及若干不太明显的设备，例如将ATM卡和存款信封传递给机器的传送器、现金给付器、收据打印机等。

考虑端口有助于测试人员定义功能性测试和系统测试所需的输入空间．类似地，输出设备提供基于输出的功能性测试信息。（例如，我们希望能够有足够的线索生成所有15个SATM屏幕。）

14.2.4 事件

事件有些复杂：既有数据方面的一些特征，又有行动方面的一些特征．事件是发生在端口设备上的系统级输入（或输出）。与数据类似，事件也可以是行动的输入和输出．事件可以是离散的（例如SATM键盘输入），也可以是连续的（例如温度、高度或压力）。离散事件必须有一定的持续时间，在实时系统中，这一点可能是至关重要的因素．我们可以把输入事件看做是破坏性读出数据，但是输出事件可以看做是破坏性写入操作。

事件与行动相似，是因为事件是现实世界物理事件和这些事件的内部逻辑表示的转换点．端口输入事件是物理到逻辑的转换，同样，端口输出事件是逻辑到物理的转换。系统测试人员应该关注事件的物理层面，而不是逻辑层面（这是集成测试人员要关注的内容）．当表示数据值内容改变物理事件的逻辑含义时，就会出现这种情况。例如在SATM系统中，当显示屏幕5时，按下B1键的端口输出事件表示"余额"，当显示屏幕6时，表示"检查"，当显示屏幕10、11和14时，表示"是"．我们把这种情况叫做"与语境有关的端口事件。"应该在每种语境中测试这种事件。

14.2.5 线索

但对于测试人员来说，线索是五种基本构造中最不经常使用的．因为要测试线索，因此测试人员通常不能在数据、事件和行动之间的交互中找出线索。线索本身出现在需求规格说明中的惟一地方，是使用快速原型法并结合场景记录器．下面就要讨论到，在控制模型中容易找出线索。这个问题的关键是，控制模型只是模型，而不是现实系统

14.2.6 基本概念之间的关系

图14-1是上述基本概念的实体/关系（E/R）模型。请注意，所有关系都是多对多的：数据和事件被概括为实体，到行动实体的两个关系用于输入和输出．同样的事件可以在多个端口上发生，一般有很多事件发生在单一的端口上．最后，一个行动可以在多个线索中发生，一个线索由多个行动合成。测试人员必须使用事件和线索保证所有五个基本概念之间的多对多关系是正确的。

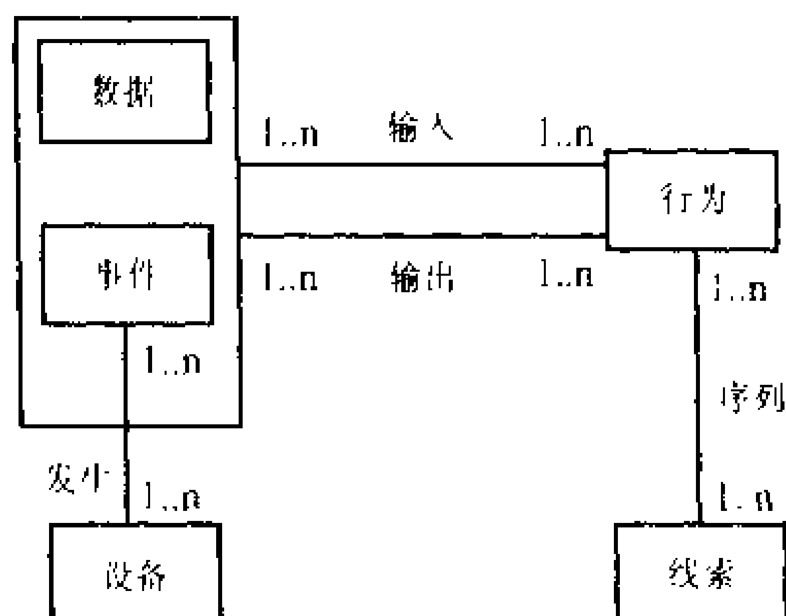


图14-1 基本概念的E/R模型

14.2.7 采用基本概念建模

各种风格的需求规格说明都要用基本概念开发系统模型。图14-2给出了三种基本形式的需求规格说明模型：结构模型、语境模型和行为模型。结构模型用于开发，表示功能分解、数据分解和组件之间的接口。语境模型常常是结构模型的开始点，强调系统设备，也比较强调行动，非常间接地关注线索。行为模型（又叫做控制模型）将五种基本构造中的四种集成到了一起。选择适当的控制模型是需求规格说明的基础：太弱的模型不能表示重要的系统行为，太强的模型一般会淹没有意思的行为。作为一般规律，决策表只对计算系统是个好选择，有限状态机对于菜单驱动的系统很好，Petri网是针对并发系统的模型。这里我们使用有限状态机描述SATM系统，第15章将使用Petri网分析线索交互。

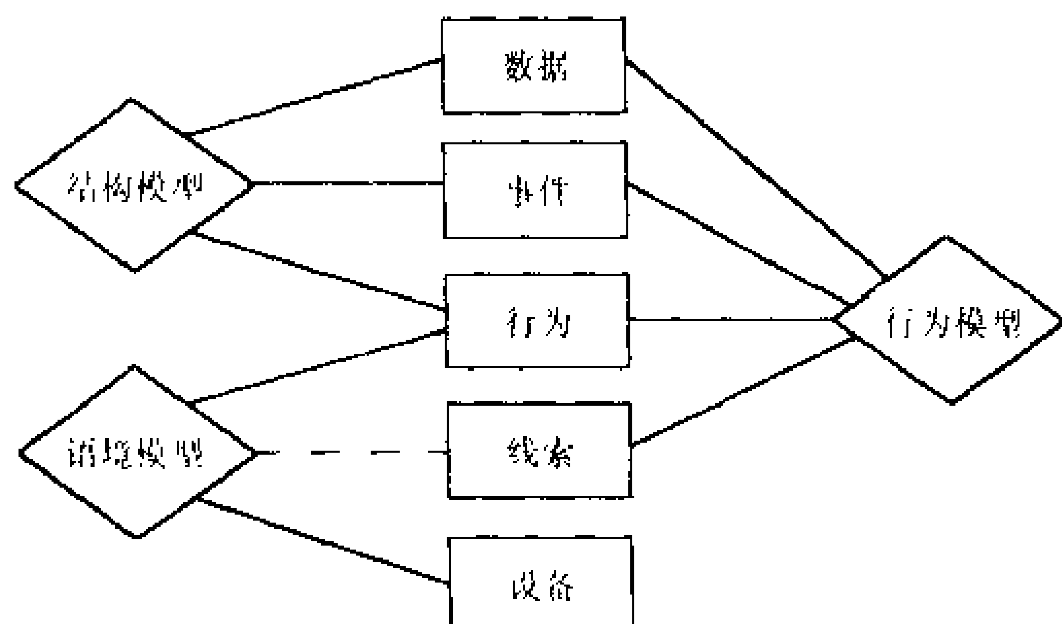


图14-2 基本结构之间的建模关系

我们必须对系统本身（现实性）和系统模型做重要的区分。考虑某个系统，其函数F只有在前提事件E1和E2发生时才会出现，这些事件能够以任意顺序出现。我们可以使用事件

划分的概念对这种情况建模，结果得到如图14-3所示的框图

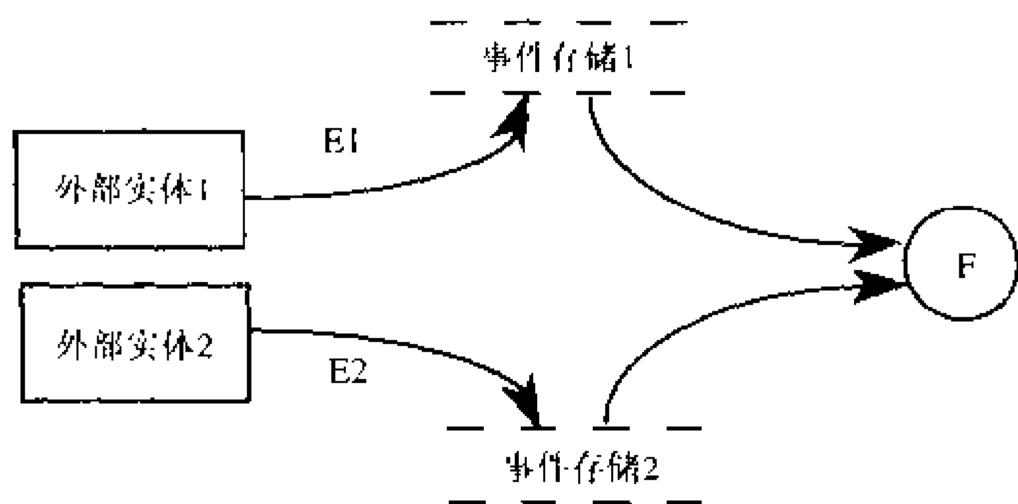


图14-3 函数F的事件划分视图

在事件划分视图中，事件E1和E2根据外部设备分别发生。当这些事件发生时，要保存在各自的事件存储中。（事件存储起毁坏性读操作的作用。）当两个事件发生后，函数F从事件存储中得到其前提信息。请注意，不能通过模型区分到底是哪个事件先发生，我们只知道两个事件一定发生。

我们还可以将系统建模为如图14-4所示的有限状态机（FSM），其中的状态记录所发生的事件。状态机视图显式地显示事件的两种顺序。

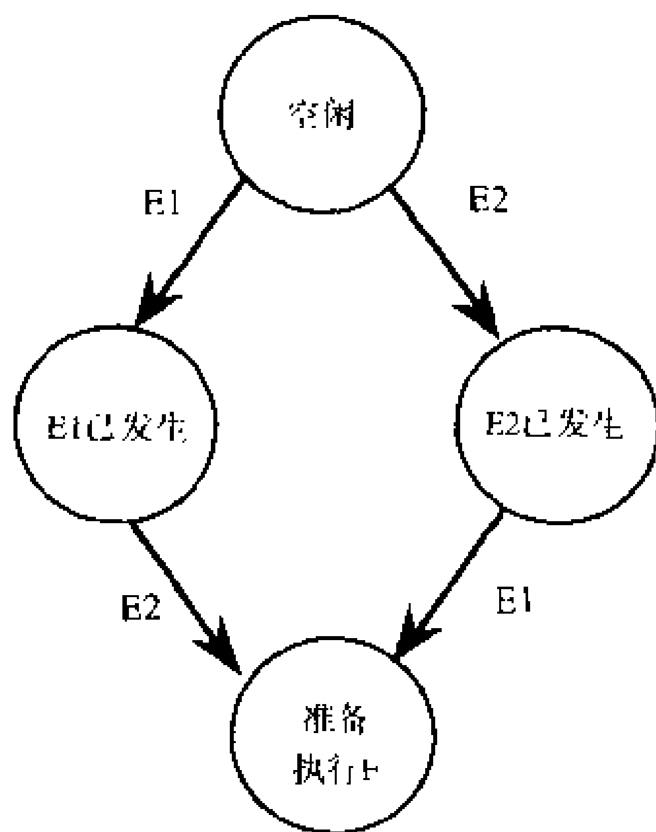


图14-4 函数F的有限状态机

两种模型都描述了函数F的相同前提，两者都不是实际系统。对于这两种模型，状态机对于测试人员来说更有用，因为路径马上可以转换为线索。

14.3 寻找线索

SATM系统的有限状态机模型，是研究系统测试线索的最佳地方。我们首先讨论状态机的层次结构，状态机的顶层如图14-5所示。在这一层，状态对应过程的阶段，转移由逻辑（而不是端口）事件引起。例如，“ATM卡输入”“状态”可分解为处理卡被塞住、卡被插反、卡传送器堵塞和对照要提供服务的ATM列表检查卡等细节的低层状态。一旦宏状态的细节通过测试，则使用某个简单线索得到下一个宏状态。

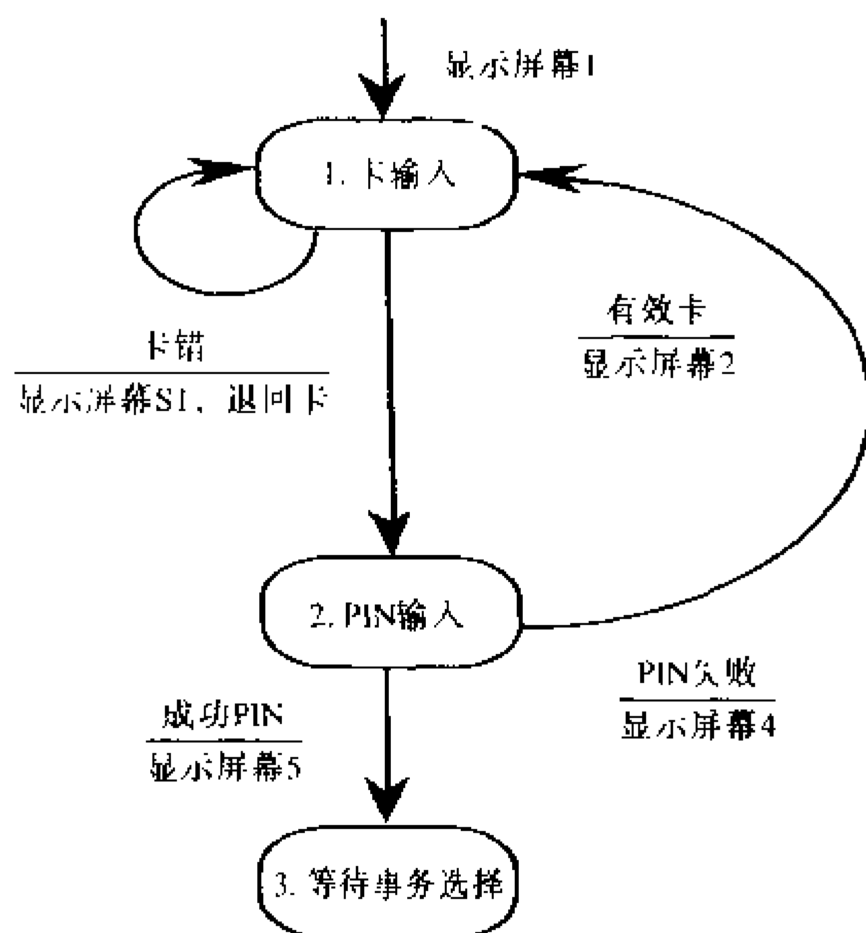


图14-5 顶层SATM状态机

“PIN输入”状态分解为如图14-6所示更详细的视图，在第12章版本的基础上稍做改进，图中还给出了邻近状态，因为这些状态是“PIN输入”部分转换的源和目的地。在这一层上，我们强调“PIN输入”机制，所有输出事件都是真正的端口事件，但是输入事件仍然是逻辑事件。状态和边的编号供稍后讨论测试覆盖时使用。

为了开始线索标识过程，我们首先列出表示状态转移的端口事件，表14-1列出了这些端口事件。我们略去退出ATM事件，因为这实际上不是PIN输入组件的一部分。

表14-1 PIN输入有限状态机中的事件

端口输入事件	端口输出事件
有效卡	显示屏幕1
卡错	显示屏幕2
正确的PIN	显示屏幕3
错误的PIN	显示屏幕4
取消	显示屏幕5

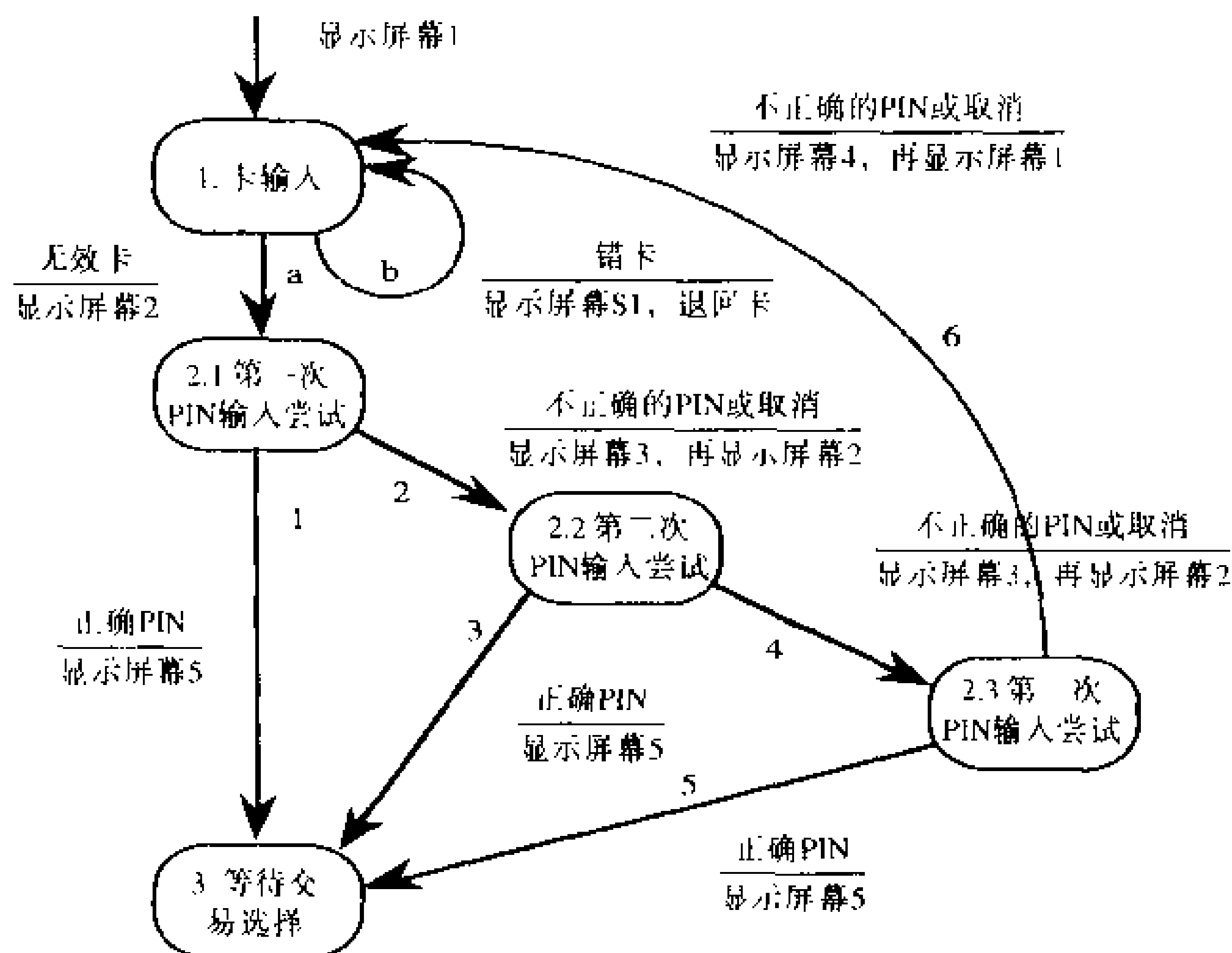


图14-6 “PIN输入”有限状态机

请注意，“正确PIN”和“错误PIN”实际上是合成端口输入事件，实际上不能输入完整的PIN，因为输入数字的任何时候都可以键入取消键。更细节的可能性如图14-7所示。真正固执的测试人员可能会将数字端口输入事件分解为实际选择（按下0、按下1……、按下9），但是这些都应该在低层测试。“PIN输入尝试”有限状态机的端口事件如表14-2所示。

表14-2 “PIN输入尝试”有限状态机中的端口事件

端口输入事件	端口输出事件
数字	回显“X———”
取消	回显“XX——”
	回显“XXX—”
	回显“XXXX”

“PIN输入”有限状态机名字中的“x”表示是通过机器的第几次尝试（第一次、第二次或第三次）。

除了“PIN输入尝试”有限状态机的真正端口事件之外，还有三个逻辑输出事件（“正确的PIN”、“错误的PIN”和“取消”），正好对应如图14-6所示的高层事件。

有限状态机的层次结构，使线索的数量成倍增长。从如图14-6所示的“第一次PIN输入尝试”状态到“等待事务处理选择”或“ATM卡输入”状态之间，有156条不同的路径。在这些路径中，有31条对应最终正确的PIN输入（第一次尝试有1条路径、第二次尝试有5条路径、第3次尝试有25条路径），其他125条路径对应错误的数字或按下取消键。这是相当典型的比例。系统的输入部分，特别是交互系统的输入部分，往往有大量线索处理输入错误和例外

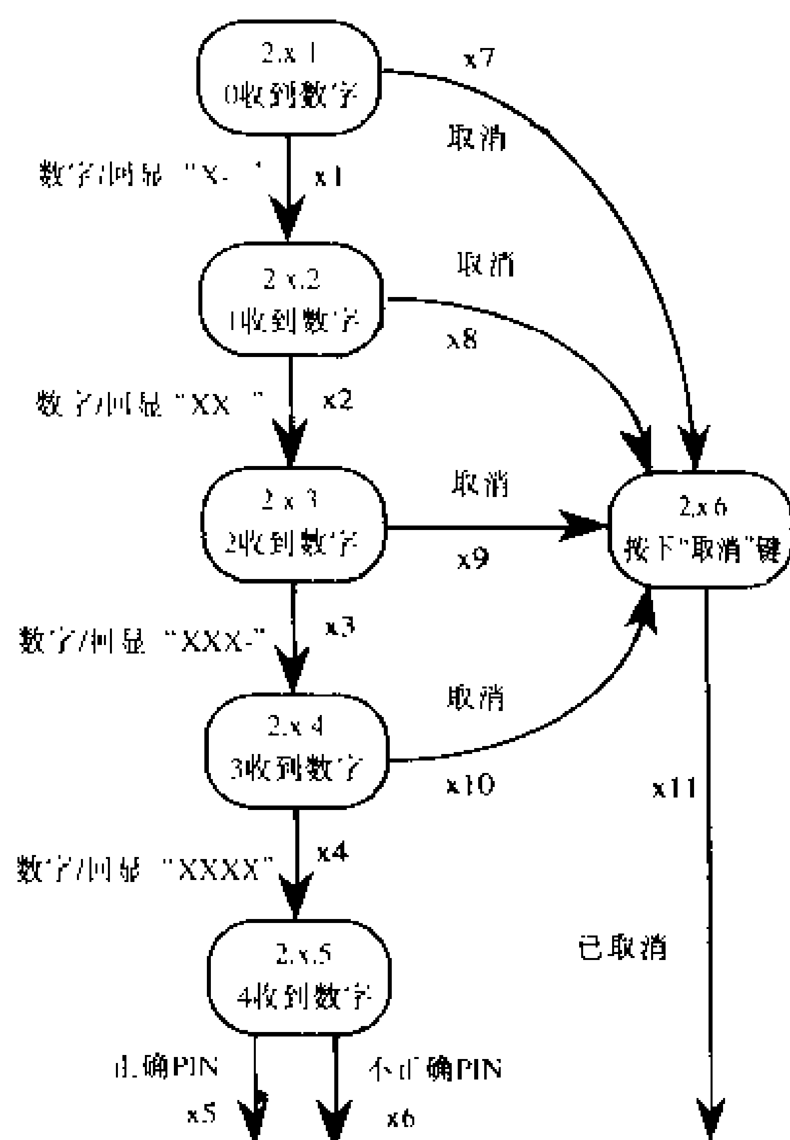


图14-7 "PIN输入尝试"有限状态机

给出状态机是一种很好的方式，其中实际端口输入事件引起转移，转移行动是端口输出事件。如果有这种有限状态机，那么为这些线索输出系统测试用例就成为机械过程，即只是追踪转移路径，并在遍历路径时注意端口输入和输出。这种交替序列是由测试执行器（人或程序）完成的。表14-3和14-4给出了走通层次状态机中两条路径的情况。表14-3对应于PIN在第一次尝试被正确输入的一个线索，表14-4对应于PIN在第一次尝试被错误输入的一个线索，在第二次尝试时输入第三个数字后，键入了取消键，到第三次才输入正确。为了使测试用例明确，我们假设PIN是1234。

表14-3 第一次PIN输入正确的端口事件序列

端口输入事件	端口输出事件
	屏幕2显示"————"
按下1	
	屏幕2显示"X———"
按下2	
	屏幕2显示"XX——"
按下3	
	屏幕2显示"XXX—"
按下4	
	屏幕2显示"XXXX"
（正确的PIN）	显示屏幕5

表14-4　第三次PIN输入正确的端口事件序列

端口输入事件	端口输出事件
	屏幕2显示"————"
按下1	
	屏幕2显示"X———"
按下2	
	屏幕2显示"XX——"
按下3	
	屏幕2显示"XXX—"
按下5	
	屏幕2显示"XXXX"
（错误的PIN）	显示屏幕3
（第二次尝试）	屏幕2显示"————"
按下1	
	屏幕2显示"X———"
按下2	
	屏幕2显示"XX——"
按下3	
	屏幕2显示"XXX—"
按下取消键	
（第二次尝试结束）	显示屏幕3
	屏幕2显示"————"
按下1	
	屏幕2显示"X———"
按下2	
	屏幕2显示"XX——"
按下3	
	屏幕2显示"XXX—"
按下4	
	屏幕2显示"XXXX"
（正确的PIN）	显示屏幕5

表14-3最后一行括号中的事件是逻辑事件，"撤回"到上级状态机，并引起到"等待事务处理选择"状态的转移。

如果仔细研究表14-3和14-4，就会发现表14-4的第三次尝试与表14-3完全一样，因此线索可以是另一个线索的子集。

14.4　线索测试的结构策略

尽管生成线索测试用例很容易，但是确定实际使用哪个测试用例这个问题更复杂（如果有自动测试执行器则不成问题。）在系统级也同样存在单元级的路径爆炸问题。与单元级采用的策略一样，我们也可以使用有向图更明智地选择要测试的线索

14.4.1 自底向上组织线索

当在层次结构中组织状态机时，可以自底向上进行。在“PIN输入尝试”状态机中，共有六条路径。如果遍历这六条路径，我们要测试三件事：正确识别并回显所输入的数字、响应取消键入和匹配预期PIN和所输入的PIN。这些路径在表14-5中描述为图14-7中的转移序列。经过路径的线索以其输入键描述，因此输入序列1234对应表14-3详细描述的线索（取消键入采用C表示）。

表14-5 “PIN输入尝试”有限状态机中的线索路径

输入事件序列	转移的路径
1234	x1、x2、x3、x4、x5
1235	x1、x2、x3、x4、x6
C	x7、x11
1C	x1、x8、x11
12C	x1、x2、x9、x11
123C	x1、x2、x3、x10、x11

一旦测试了这个部分之后，可以上升到“PIN输入尝试”状态机，共有四条路径。这些路径对应三次尝试机制以及向用户显示的屏幕序列。在表14-6中，PIN输入状态机中的路径（如图14-6所示）用转移序列命名。

表14-6 “PIN输入尝试”有限状态机中的线索路径

输入事件序列	路径转移
1234	1
12351234	2，3
1235C1234	2，4，5
CCC	2，4，6

这些线索采用心中的路径遍历目标标识。在讨论结构性测试时曾经提到过，这些目标可能产生误导。假设是路径遍历会发现缺陷，遍历各种路径会产生冗余。表14-6中的最后路径说明结构目标会产生怎样的问题。按下取消键三次确实使三次尝试机制失败，并将系统返回到“ATM卡输入”状态，但是这看起来是退化线索。这些线索还有一个更严重的问题：由于是分层状态机，因此我们实际上不能单独执行。对于表14-5中的1235输入序列实际上会发生什么呢？要经过“PIN输入尝试”状态机的有关路径，然后“返回”到“PIN输入”状态机上，在这个状态机上看起来像逻辑事件（错误的PIN），引起到状态2.2的转移（第二次“PIN输入尝试”）。如果没有按下其他键，则这个状态机仍然留在状态2.2中。以下讨论如何解决这种问题。

14.4.2 节点与边覆盖指标

由于有限状态机是有向图，因此可以使用与单元级相同的测试覆盖指标。层次结构关系意味着上层状态机必须把下层状态机作为输入和返回过程处理。（实际上，我们需要在再一层上做这个工作，以得到以“ATM卡输入”状态开始的真正线索。）两种明显的选择是节点覆盖和边覆盖。表14-7扩展了表14-4，以给出三次尝试线索的节点和边覆盖。

表14-7 一个线索的节点和边遍历

端口输入事件	端口输出事件	节点	边
	屏幕2显示“————”	2.1	a
按下1		2.1.1	
	屏幕2显示“X———”		x1
按下2		2.1.2	
	屏幕2显示“XX——”		x2
按下3		2.1.3	
	屏幕2显示“XXX—”		x3
按下5		2.1.4	
	屏幕2显示“XXXX”		x4
（错误的PIN）	显示屏幕3	2.1.5，3	x6，2
（第二次尝试）	屏幕2显示“————”	2.2	
按下1		2.2.1	
	屏幕2显示“X———”		x1
按下2		2.2.2	
	屏幕2显示“XX——”		x2
按下3		2.2.3	
	屏幕2显示“XXX—”		x3
按下取消键		2.2.4	x10
（第二次尝试结束）	显示屏幕3	2.2.6	x11
	屏幕2显示“————”	2.3	4
按下1		2.3.1	
	屏幕2显示“X———”		x1
按下2		2.3.2	
	屏幕2显示“XX——”		x2
按下3		2.3.3	
	屏幕2显示“XXX—”		x3
按下4		2.3.4	
	屏幕2显示“XXXX”		x4
（正确的PIN）	显示屏幕5	2.3.5，3	x5,5

节点（状态）覆盖可与单元级的状态覆盖类比。在PIN输入例子中，可以不通过执行输入正确PIN的线索得到代码覆盖。如果观察表14-8就会看到两个线索（由C1234和123C1C1C引起）遍历两个状态机中的所有状态。

表14-8 线索/状态关联

输入事件	2.1	2.x.1	2.x.2	2.x.3	2.x.4	2.x.5	2.2.6	2.2	2.3	3	1
1234	x	x	x	x	x	x				x	
12351234	x	x	x	x	x	x		x		x	
C1234	x	x	x	x	x	x	x	x		x	
1C12C1234	x	x	x	x			x	x	x	x	
123C1C1C	x	x	x	x	x		x	x	x		x

边（状态转换）覆盖是更可接受的标准。如果状态机是“构建良好”的（通过端口事件转换），则边覆盖也可以保证端口事件覆盖。表14-9中的线索是按结构挑选的，以保证较少重复经过的边（由于按下取消键引起）。

表14-9 线索/转换关联

输入事件	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	1	2	3	4	5	6
1234	x	x	x	x	x							x					
12351234	x	x	x	x	x	x							x	x			
C1234	x	x	x	x	x		x				x		x	x			
1C12C1234	x	x	x	x	x			x	x		x		x		x	x	
123C1C1C	x	x	x					x		x	x		x		x		x

14.5 线索测试的功能策略

基于有限状态机标识线索的方法显然很有用，但是如果被测系统没有行为模型怎么办呢？测试工艺师有两种选择：开发行为模型，或进行功能性测试在系统级上类比。在标识功能性测试用例时，我们曾经使用了输入和输出空间信息，以及功能本身。这里采用通过三个基本概念（事件、端口和数据）导出的覆盖指标描述功能线索。

14.5.1 基于事件的线索测试

考虑端口输入事件的空间。有五个端口输入事件覆盖指标比较重要，达到这些级别的系统测试覆盖要求提供一组线索，使得：

PI1：每个端口输入事件发生。

PI2：端口输入事件的常见序列发生。

PI3：每个端口输入事件在所有“相关”数据语境中发生。

PI4：对于给定语境，所有“不合适”的输入事件发生。

PI5：对于给定语境，所有可能的输入事件发生。

PI1是最低限度指标，对于大多数系统都做不到。PI2覆盖是最常见的，对应于系统测试

的直觉观点，因为它处理的是"一般使用"，但是很难量化。什么是输入事件的常见序列？什么是输入事件的不常见序列？

后三种指标采用"语境"定义。最好把语境看做是事件静止点。在SATM系统中，屏幕显示发生在事件静止点上。PI3指标处理与语境有关的端口输入事件。这些是物理输入事件，具有由发生这些事件语境确定的逻辑含义。例如在SATM系统中，按下B1功能键在五种单独的语境中发生（屏幕显示），并具有三种不同的含义。这种指标的关键是，事件要在所有其语境中驱动。PI4和PI5指标常常被试图分解系统的测试人员用做一种非形式化的基础。在给定语境中，要提供没有预料到的输入事件看看会出现什么情况。例如在SATM系统中，在"PIN输入"过程中按下功能键会出现什么情况？合适的事件是按下数字和取消键，不合适的输入事件是按下B1、B2和B3键。

这有一部分属于规格说明问题：我们在讨论规定行为（应该发生的事）和禁止行为（不应该发生的事）之间的差别。大多数需求规格说明只尽力描述规定行为，通常由测试人员发现规定行为。负责维护我住宅附近的ATM系统的设计人员告诉我，有一次有人竟向存款信封槽塞入鱼肉三明治。（显然他们把ATM机当成垃圾箱了。）银行所有人无论如何也不会想到将插入鱼肉三明治作为一种端口输入事件。PI4和PI5指标通常非常有效，但是会产生一种很奇怪的困难。测试人员怎么知道对于规定行为输入预期响应是什么呢？系统只是忽略这些输入吗？是否应该输出警告消息？通常这些问题的答案要靠测试人员的直觉确定。如果时间允许，这是对需求规格说明反馈的很好时机。对于快速原型法或可执行规格说明来说，这些方面也是重点。

我们还可以根据端口输出事件定义两种覆盖指标：

PO1：每个端口输出事件发生。

PO2：每个端口输出事件在每种原因下发生。

PO1覆盖是可接受的最低限度，对于有大量错误条件输出消息的系统尤其有效。（SATM系统没有太多的输出消息。）PO2覆盖是一种好的目标，但是很难定量描述。第15章在研究线索交互问题时还会讨论这个问题。现在请注意，PO2覆盖是指与端口输出事件有关的交互线索。通常给定输出事件只有少量原因。在SATM系统中，有三种原因会显示屏幕时10：终端储备的现金可能用光，可能要连接到中央银行获得账户余额，取款通道可能阻塞。在实践中，现场问题报告发现的最困难的问题是由没有被怀疑到的原因引起的输出。例如：我住宅附近的ATM系统（不是SATM）有一个屏幕显示"已经达到每日取款最高限额"。当天内取款金额超过300美元时就会出现这个屏幕。当我看到这个屏幕时，曾经认为我妻子取出大笔钱（线索交互），因此我请求取少一点现金。我后来发现只要ATM系统给付器中的现金不多时，也会出现这个屏幕。银行不是向先来的用户提供大量现金，而是宁愿为更多用户提供少量现金。

14.5.2 基于端口的线索测试

基于端口的测试是基于事件测试的有用补充。通过基于端口的测试，对于每个端口都要询问端口上会出现什么事件。然后根据每个端口的事件列表寻找使用输入端口和输出端口的线索（假设已经定义了这种事件列表，有些需求规格说明技术要求提供这种列表）。基于端口的测试对于端口设备来自外部提供商的系统特别有用。基于端口测试的主要理由可以从基本构造的E/R模型中看出（如图14-1所示）。设备和事件之间的多对多测试应该在两个方向上进行。基于事件的测试覆盖从事件到端口的一对多关系，反之，基于端口的测试覆盖从端口到事件的一对多关系。SATM系统不能使用这种测试，因为SATM不发生在多个端口上。

14.5.3 基于数据的线索测试

基于端口和基于事件的测试适合主要以事件驱动的系统，这种系统有时被叫做“反应式”系统，因为这些系统要对激励（端口输入事件）做出响应，并且反应常常以端口输出事件的形式做出。反应式系统有两个重要特征：一是长时间运行（与完成短时间大量计算的工资处理程序相反），二是维持与环境的关系。一般地，事件驱动的反应式系统没有有意思的数据模型（例如SATM系统中的数据模型），因此基于数据模型的线索不是特别有用。那么数据驱动的传统系统会怎样呢？这些在Topper（1993）中被叫做“静态”的系统是转换式（而不是反应式）系统，支持以数据库为基础的事务处理。一般采用E/R模型描述这些系统，因此有大量系统测试线索来源。为了使讨论与大家熟悉的内容联系起来，我们使用简单图书馆系统的E/R模型，图14-8来自Topper（1993）。

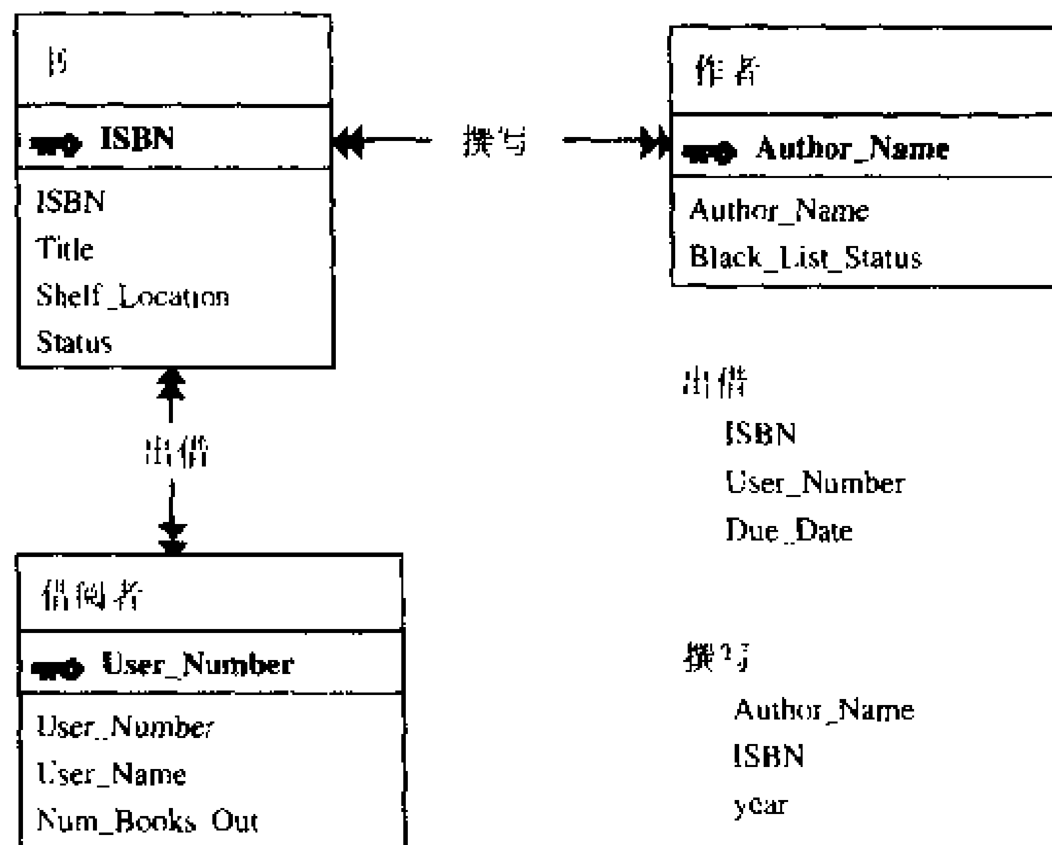


图14-8 图书馆的E/R模型

以下是图书馆系统的一些典型事务处理：

1. 图书馆添加图书。
2. 图书馆撤除图书。
3. 图书馆增加借阅者。
4. 图书馆删除借阅者。
5. 向借阅者出借图书。
6. 借阅者返还图书处理

这些事务都是主线线索。事实上，这些事务表示的是线索族。例如，假设当前借阅图书数量已经达到限额的借阅者试图办理图书外借数量，这就是一个很好的边界值例子。还可以测试返还不归该图书馆拥有的图书，再比如删除手头还有一些图书未还的借阅者。所有这些都是值得测试的线索，并且都是系统级的线索。

通过仔细研究实体/关系模型中的信息，可以找出所有这些例子，还有很多其他线索。与基于事件的测试一样，我们通过基于数据的覆盖指标描述线索集合。这些指标与关系关联是有重要原因的，关系中的信息一般包含在系统级线索中，而实体中的线索一般局限于单元级。（当把E/R模型作为面向对象分析的切入点时，这一点会由封装保证。）

DM1：检查每个关系的基数。

DM2：检查每个关系的参与。

DM3：检查关系之间的函数依赖关系。

基数指第3章曾经讨论过的关系的四种可能性：一对一、一对多、多对一和多对多。在图书馆例子中，借出和撰写关系是多对多关系，说明一位作者可以撰写多本书，并且一本书可以有多位作者；一本书可以借给很多借阅者（顺序借出），一位借阅者可以借阅多本书。这些可能性都会产生有用的系统测试线索。

参与指实体的每个实例是否参与到关系中。在撰写关系中，Book（书）和Author（作者）实体都要强制参与（不能有没有作者的图书，也不能没有图书作品的作者）。在有些建模手段中，用数字限额表示参与。例如，作者实体可以表示为“最少1位最多12位”。如果通过了这种信息，则可以直接产生很明显的边界值系统测试线索。

有时事务处理确定关系之间的显式逻辑联系，这种逻辑联系叫做函数依赖关系。例如，不能借出不属于图书馆的图书，也不能删除已经外借的图书。此外，还不能删除手头还有图书的借阅者。如果数据库被规范化，则这类依赖关系会减少，但是依然存在，并且可以用来产生有意思的系统测试线索。

14.6 SATM测试线索

如果把本章讨论的内容运用到SATM系统中，则可以得到能够进行贯穿系统级测试的一组线索。我们通过一种全状态模型开发这组线索。在全状态模型中，状态表示关键原子系统功能。宏级状态是："ATM卡输入"、"PIN输入"、"事务处理请求"（与处理）和"会话管理"。状态顺序就是测试顺序，因为这些阶段是以前提顺序给出的。（只有在成功地输入ATM卡之后才能输入PIN，只有成功输入PIN之后才能请求事务处理，等等。）我们还需要一些定义带有PAN、预期PIN和账户余额的实际账户前提数据。这些数据在表14-10中给出。两种不太明显的前提是，ATM终端初始显示屏幕1，提取现金给付器的总现金为500美元（面值10美元）。

表14-10 SATM测试数据

PAN	预期PIN	支票余额（美元）	储蓄余额（美元）
100	1234	1000.00	800.00
200	4567	100.00	90.00
300	6789	25.00	20.00

我们采用表格的形式描述线索，一对行对应四个主要阶段中的每一个阶段的端口输入和预期端口输出。先讨论三个基本线索，每种线索对应一种事务处理类型（余额查询、存款和取款）。

在线索1中，输入了PAN = 100的有效ATM卡，导致显示屏幕2。输入PIN数字1234，由于这个数字与该PAN预期PIN匹配，因此显示使客户能够选择事务处理类型的屏幕5。当第一次按下B1键（要求余额查询）时，会显示询问账户的屏幕6。当第二次按下B1键（支票）时，会显示屏幕14，并将支票账户余额（1000.00美元）打印到收据上。当按下B2键时，会显示屏幕15，打印收据，退出ATM卡，显示屏幕1。

线索1（余额）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	1234	B1，B1	B2
端口输出	屏幕2	屏幕5	屏幕6，屏幕14，1000.00美元	屏幕15，退出ATM卡，屏幕1

线索2是向支票账户中存款：同样的PAN和PIN，但是当显示屏幕5时按下B2。当显示屏幕6时按下B1键。当显示屏幕7时输入金额25.00，这时显示屏幕13。存款通道打开，存款信封放在存款槽中，显示屏幕14。当按下B2键时，显示屏幕15，将新的支票账户余额1025.00美元打印在收据上，然后退回ATM卡，显示屏幕1。

线索2（存款）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	1234	B2，B1，25.00	B2
端口输出	屏幕2	屏幕5	屏幕6，屏幕7，屏幕13，存款通道打开，屏幕14，1025.00美元	屏幕15，退出ATM卡，屏幕1

线索3是从储蓄账户中取款：同样的PAN和PIN，但是当显示屏幕5时按下B3，当显示屏幕6时按下B2键，当显示屏幕7时输入金额30.00，这时显示屏幕11，取款通道打开，送出三张10美元钞票，显示屏幕14，当按下B2键时，显示屏幕15，将新的储蓄账户余额770.00美元打印在收据上，然后退回ATM卡，显示屏幕1

线索3（取款）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	1234	B2、B2、30.00	B2
端口输出	屏幕2	屏幕5	屏幕6、屏幕7、屏幕11、取款通道打开、三张10美元钞票、屏幕14、770.00美元	屏幕15、退出ATM卡、屏幕1

需要给出几个这种详细描述的线索，以显示模式。剩下的线索通过作为测试线索目标的输入和输出事件描述。

线索4是SATM系统中最短的线索，包含无效的ATM卡，立即被退出

线索4	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	400			
端口输出	退出ATM卡，屏幕1			

继续给出线索1的宏状态，下面进行各种“PIN输入”。通过产生“PIN输入”有限状态机的边覆盖的表14-9，可得到四个新线索。

线索5（余额）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	12351234	与线索1相同	
端口输出	屏幕2	屏幕3，2，5		

线索6（余额）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	C1234	与线索1相同	
端口输出	屏幕2	屏幕3，2，5		

线索7（余额）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	1C12C1234	与线索1相同	
端口输出	屏幕2	屏幕3，2，3，2，5		

线索8（余额）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	123C1C1C		
端口输出	屏幕2	屏幕3，2，3，2，4，1		

转到“事务处理请求”阶段，根据事务处理的类型（余额、存款或取款），账户（支票或储蓄）以及所请求金额的几种不同处理，有不同情况。线索1、2和3覆盖了各种事务处理类型和账户，因此以下重点讨论金额驱动的线索。线索9拒绝提取非10美元整数倍现金的尝试，线索10拒绝提取现金数额多于账户余额的尝试，线索11拒绝提取现金数额大于给付器现

有现金的尝试。

线索9（提取）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	1234	B3，B2，15.00，取消	B2
端口输出	屏幕2	屏幕5	屏幕6，7，9，7	屏幕15，退出ATM卡，屏幕1

线索10（提取）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	300	6789	B3，B2，50.00，取消	B2
端口输出	屏幕2	屏幕5	屏幕6，7，8	屏幕15，退出ATM卡，屏幕1

线索11（提取）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	1234	B3，B2，510.00，取消	B2
端口输出	屏幕2	屏幕5	屏幕6，7，10	屏幕15，退出ATM卡，屏幕1

研究了事务处理部分之后，下面转到会话管理阶段，要测试多事务处理部分。

线索12（余额）	ATM卡输入（PAN）	PIN输入	事务处理请求	会话管理
端口输入	100	1234	B1，B1	B2，取消
端口输出	屏幕2	屏幕5	屏幕6，屏幕14，1000.00美元	屏幕15，屏幕5，屏幕15，退出ATM卡，屏幕1

到此为止，线索提供对除屏幕12之外的所有输出屏幕的覆盖，屏幕12告诉用户存款不能被处理，产生这种条件是有问题的（可能应该将鱼肉三明治塞入存款信封槽中），这是一种由硬件失效前提选择的线索，我们在这里给出线索名称，即线索13。下面我们开发线索14~22，检验与语境有关的输入事件，如表14-11所示。请注意，头13个线索中的一些也是与语境有关的。

表14-11 与语境有关的输入事件线索

线索	按下的键	屏幕	逻辑含义
6	取消	2	PIN输入错误
14	取消	5	事务处理选择错误
15	取消	6	账户选择错误
16	取消	7	金额选择错误
17	取消	8	金额选择错误
18	取消	13	取款信封未就绪
1	B1	5	余额
1	B1	6	支票
19	B1	10	是（非取款事务处理）
20	B1	12	是（非存款事务处理）
12	B1	14	是（另一个事务处理）
2	B2	5	存款
3	B2	6	储蓄
21	B2	10	否（没有其他事务处理）
22	B2	12	否（没有其他事务处理）
1	B2	14	否（没有其他事务处理）

这22个线索构成已经描述过的SATM系统部分的合理测试。当然，有些方面还没有测试，比如账户余额。请考虑两个线索，一个在账户中存入40美元，另一个取出80美元，并假设“ATM卡输入”阶段从中央银行得到的余额是50美元。有两种可能：一种是使用中央银行的余额，记录所有事务处理，然后再每日向中央银行汇总所有处理。另一种可能是维护本地流转余额，即在请求事务中要显示的余额。如果使用中央银行的余额，则应该拒绝上述取款尝试，如果使用本地余额，则可以取款。这种细节在规格说明中没有描述。第15章在研究线索交互时还会讨论这个问题。

SATM系统另一种明显未测试的部分，是屏幕7和8出现的“金额输入”过程。在金额输入过程中的任何时候都可能输入取消键，这会产生比“PIN输入”还要多的变化情况。对于“金额输入”可以使用一种更微妙（因此更有意义）的测试。当输入金额后实际会发生什么？更具体地说，假设输入40美元。我们期望对于每个数字键入都有一次回显，但是回显出现在什么位置上呢？显然有两种解决方案：永远要求输入6位数字（因此应该输入“004000”），或先显示高位数字，当输入后续数字时向左移位，如图14-9所示。大多数ATM系统都使用移位方法，这会产生微妙问题：ATM系统如何知道所有金额数字都已经输入呢？ATM系统显然不能预测存款金额是40美元，而不是400或4000美元，因为当最后数字被输入后，没有“输入”键可以用来指示输入结束。指出这种问题，是因为这是测试人员发现需求规格说明常常疏漏的细节问题的好例子。（这类细节问题有可能通过快速原型法或使用可执行规格说明发现。）

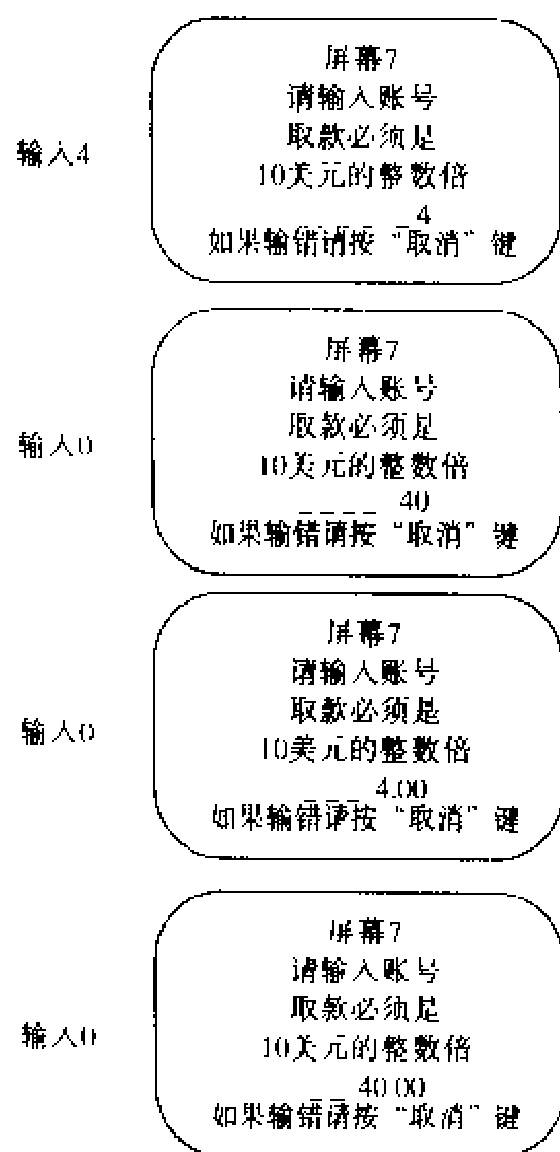


图14-9　向左移位数字回显

14.7 系统测试指导方针

如果不允许复合会话（即多个事务），并且不考虑“金额输入”的多样性，则对于SATM系统中的每个有效账户，共有435个不同的线索，如果考虑复合会话和“金额输入”多种可能性，则SATM系统会有数万个线索。本章最后讨论解决线索爆炸问题的三种策略

14.7.1 伪结构系统测试

在讨论单元测试时，曾经介绍过通过功能和结构性测试相结合，产生我们所希望的交叉检查效果。对于系统级线索也有类似问题：第14.5节定义了10个系统级功能覆盖指标，第14.4节定义了两个基于图的指标（节点和边覆盖）。可以像在单元级上使用DD-路径标识功能性测试用例中的漏洞和冗余一样，将基于图的指标用做功能线索的一种交叉检查。我们只能说明伪结构性测试（Jorgensen，1994），因为节点和边覆盖指标都以系统的控制模型定义，不是直接从系统实现中导出的。（本章开始时曾经讨论过现实和现实模型之间的差别。）一般来说，行为模型只是系统实际情况的近似，这也是为什么我们可以将模型多层细分的原因。如果建立一种真正的结构模型，那么这样模型会太大、太复杂，很难使用。伪结构指标的一个很大的弱点是，下层模型可能是差的选择。三种最常见的行为模型（决策表、有限状态机和Petri网）分别适合转换式、交互式和并发系统

决策表和有限状态机是原子系统功能性测试的很好选择。如果原子系统功能采用决策表描述，则条件一般会包括端口输入事件，行动就是端口输出事件，然后可以设计覆盖每个条件、每个行动，或更完备地说，每个规则的测试用例。我们已经讨论过有限状态机模型，测试用例可以覆盖每个状态、每种转换和每条路径。

基于决策表的线索测试是很麻烦的，可以根据不同决策表的规则顺序描述线索，但是这样采用覆盖指标跟踪很不清晰。有限状态机是最基本的选择，如果存在某种形式的交互，则Petri网是最佳选择，这时可以设计覆盖所有地点、每种转移和所有转移序列的线索测试。

14.7.2 运行剖面

对于大多数一般形式，齐夫定律（Zipf's Law）都成立，即80%的活动发生在20%的空间中，活动和空间可以有很多方式解释：桌子上乱七八糟的人、桌子上的大部分东西都很难用到；对于自己所喜欢的程序设计语言，程序员所使用的功能很少超过其总功能的20%；莎士比亚（其作品包含大量词汇）绝大多数情况只使用很小一部分词汇。齐夫定律在多个方面都适用于软件（和测试）。齐夫定律对于测试人员最有用的解释是，空间由所有可能的线索组成，活动是线索的执行（或遍历）。因此，对于有很多线索的系统，80%的执行只遍历20%的线索。

前面提到过当执行缺陷时失效发生的情况，测试的整体思想是执行测试用例，使得当失

效发生时，可发现缺陷的存在。我们可以做出重要区分：系统中的缺陷分布只是间接地与系统的可靠性有关。系统可靠性最简单的观点是，在特定的事件区间内没有失效发生的概率。（请注意，这里甚至没有涉及缺陷、缺陷数量或缺陷密度。）如果仅有的缺陷只“局限”在很少遍历的线索中，则与同样数量缺陷存在于“经常使用”的线索中相比，整体可靠性较高。运行剖面的思想，是确定各种线索的执行频率，并使用这种信息为系统测试选择线索。尤其当测试时间很紧（这种情况很常见）时，操作剖面能够使在最常遍历的线索中引入失效的线索的发现概率最大化。

确定系统运行剖面的一种方法是使用决策树。如果系统行为采用层次状态机建模，例如前面针对SATM系统所建的模型，则决策树的效果更好。对于任何状态，我们找出（或估计）每种向外转移的概率（各种转移概率的和必须是1）。当状态分解为下层状态时，低层的概率是上层概率的“细分”。图14-10给出了假想转移概率的结果。给定转移概率，线索的总概率就是线索各转移概率的积。表14-12给出了最高和最低频率线索的计算结果。

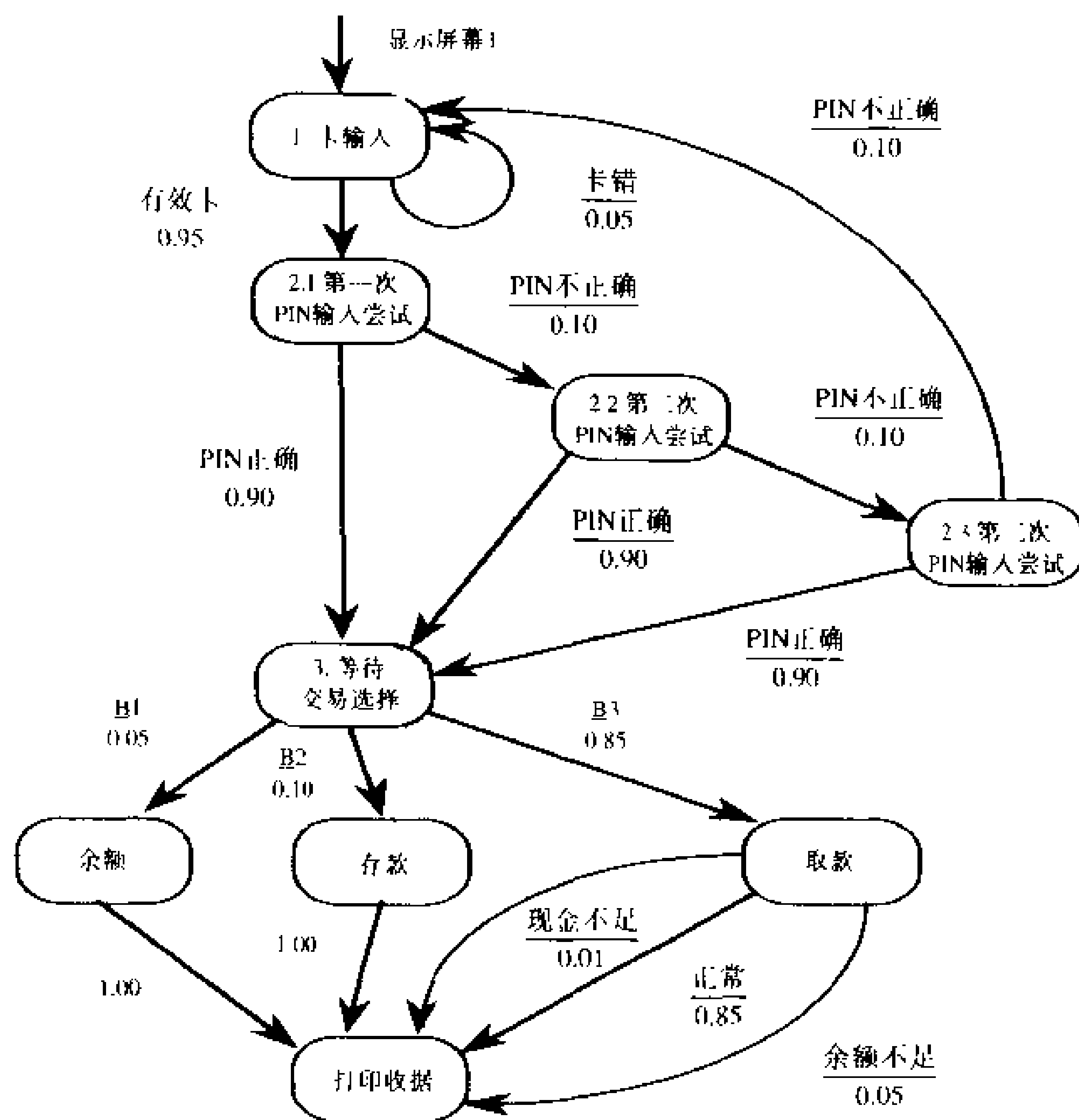


图14-10　SATM系统的转移概率

表14-12 线索概率

常见线索	概率	罕见线索	概率
有效ATM卡	0.95	有效ATM卡	0.95
PIN第一次尝试正确	0.90	PIN第一次尝试错误	0.10
取款	0.85	PIN第二次尝试错误	0.10
正常	0.85	PIN第三次尝试正确	0.90
		取款	0.85
		现金不足	0.01
	0.6177375		0.00072675

运行剖面提供所交付系统的使用情况，其意义不仅在于优化系统测试。这些剖面还可以结合模拟器使用，以得到执行时间性能和系统事务处理能力的早期指示。很多时候，客户是收集系统使用情况信息的好来源，因此这种系统测试方法常常被接受，因为这种方法试图复现被交付系统的实际情况。

14.7.3 累进测试与回归测试

第12章在讨论软件开发生命周期时，曾经提到使用构建引入累进测试需求。当构建2加到构建1上时，要测试构建2中的新内容，还要重新测试构建1，以确定新内容不会对构建1的内容产生有害的影响。（这种由于对现有系统修改而在系统中引入新缺陷的"波及效应"在业界的平均出现概率是20%。）如果项目有多个构建，回归测试意味着测试的大量重复，尤其是早期构建。通过研究累进测试和回归测试之间的差别可减少这类重复。

回归测试的最常见方法是直接重复系统测试。通过根据回归测试和伴随测试目标选择测试线索，可以改进这种方法（并大大降低工作量）。对于伴随测试，我们测试的是新内容，因此预期失效率要比回归测试的失效率高。另一种区别是，由于我们期望通过伴随测试复现更多缺陷，需要定位这些缺陷，要求测试用例具有诊断能力，也就是说，测试只能以几种方式失效。对于基于线索的测试，伴随测试应该使用更短的只能以几种方式失效的线索。这些线索可以按我们在讨论SATM线索测试集合时介绍过的方式组织，使得较长的线索由较短的（以前已通过测试的）线索组成。

回归测试的预期失效数较少，也不太关心缺陷定位。将两个方面综合起来，这意味着回归测试应该使用更长的线索，可以以多种方式失效。如果从覆盖角度考虑，累进测试和回归测试都需要大的覆盖，但是覆盖密度是不同的。对于累进测试线索来说，状态和转移覆盖指标（例如表14-8和14-9给出的指标）是稀疏的，对于回归测试线索来说，是紧密的。这与运行剖面的使用有一定的对立。作为一种规则，"好的"回归测试线索会有低的运行频率，伴随测试线索具有高的运行频率。

14.8 参考文献

Topper, Andrew et al., *Structured Methods, Merging Models, Techniques, and CASE*, McGraw-Hill, New York, 1993.

Jorgensen, Paul C., An operational common denominator to the structured real-time methods, *Proceedings of the Fifth Structured Techniques Association (STA 5) Conference*, Chicago, May 11, 1989.

Jorgensen, Paul C., System testing with pseudo-structures, *Amer Programmer*, Vol. 7, No 4, pp. 29–34, April 1994.

14.9 练习

1. 系统测试，特别是交互式系统的系统测试中的一个问题是，预测用户可能做出的各种奇怪操作。如果SATM系统的客户输入三位PIN数字并离开，会出现什么情况呢？

2. 为了控制异常用户行为（行为是不正常的，用户并没有异常），SATM系统可能要引入有30秒超时的计时器。如果在30秒内没有端口输入事件，则SATM系统会询问用户是否需要更多时间，用户可以回答是或否。请设计新屏幕，并标识实现这种超时事件的端口事件。

3. 假设在SATM系统中增加了超时功能，应该执行什么回归测试？

4. 请对“PIN输入尝试”有限状态机（如图14-6所示）做进一步改进，以实现超时功能，然后再修改表14-3给出的线索测试用例。

5. 文本说明“B1功能键在五种不同语境中出现（屏幕显示），并具有三种不同含义。”请研究15个屏幕（事件静止点），并确定按下B1键是否具有三种或五种不同的逻辑含义。

6. 结合运行剖面使用覆盖指标是否有意义？请讨论。

第15章

交互测试

由于交互引起的缺陷和失效给测试人员带来棘手问题。这种缺陷和失效的微妙性很难发现，通过测试发现甚至更困难。这些都是很深层次的缺陷，即使经过大量的线索测试，仍然会留在系统中。但是，已交付系统经过一段时间的运行之后，常常出现交互缺陷。在一般情况下，交互缺陷的执行概率很低，只有在执行大量线索之后才会出现。本章大部分篇幅将专门介绍交互形式，而不是如何测试。因此本章更关注的是需求规格说明，而不是测试。这种联系是很重要的：知道如何描述交互，是检查和测试交互的第一步。本章还要就交互缺陷和失效在哲学和一般数学层次上进行一些讨论，不能期望测试自己不理解的东西，本章首先对已经讨论过的五种基本构造做重要补充，并使用这些构造定义交互分类。然后本章对传统Petri网做简单扩充，以反映这些基本构造，并利用简单自动柜员机（SATM）和土星牌挡风玻璃雨刷系统例子贯穿整个讨论，偶而也使用电话系统的例子。本章最后将所提出的分类运用于重要的应用程序：客户-服务器系统。

15.1 交互的语境

描述和测试交互很困难的部分原因，是因为交互太常见了。日常生活中有那么多的交互事物：人员、汽车驾驶员、规则、化学品合成与提取等。我们关心的是软件控制系统中的交互问题（特别是未预料的交互），因此把本章对交互的讨论局限在基本系统构造中：行动、数据、事件、端口和线索。

建立交互语境的一种方式，是把交互看做五种构造之间的关系。如果采用这种观点，就会发现InteractsWith关系是一种每个实体上的自反关系（数据与数据交互、行动与行动交互，等等）。InteractsWith关系还是数据与事件、数据与线索以及事件与线索之间的二元关系。但是数据建模并不是死胡同。只要数据模型包含这种普遍存在的关系，就说明遗漏了重要实体。如果在相当抽象的构造中增加一些现实因素，就可以得到更有用的交互研究框架。遗漏元素是定位，而定位有两个成份：时间和位置。数据建模提供另一种选择：我们可以把定位当做第六种基本实体，或作为其他五种实体的一种属性。这里选择当做属性。

将定位（时间和位置）作为所有五种基本构造的属性意味着什么？这实际上是几乎所有需求、规格说明表示法和技术的缺点。（这可能也是交互很少被认识和测试的原因。）有关定位的信息通常在系统实现时创建。有时定位信息被强制要求在需求中提供，如果是这种情况，则需求实际上是强制实现选择。以下首先澄清定位的两种成份：时间和位置。

我们可以从两个方面认识时间：作为一种瞬时的时间，或一种持续的时间。瞬时观点使

我们能够描述事物发生的时间，它是时间轴上的一个点。持续观点是时间轴上的一个区间。当考虑持续时间时，通常对时间区间的长度感兴趣，而不是端点（开始和结束时间）。两种观点都很有用，因为线索执行时都有持续长度，执行时也都有时间点。类似的观察也适用于事件。事件常常有很短的持续时间，如果持续时间太短，以至于事件没有被系统意识到，就会产生问题。

位置方面的问题比较容易，可以对位置采用非常实用的实际观点，并采用某种协同系统描述。位置可以是有原点的三维笛卡儿坐标系，也可以是经纬度地理位置。对于大多数系统，将位置稍微抽象一点，用处理器空间表示会更方便。总之，时间和位置会告诉测试人员什么时间、什么地点发生的某事，这是理解交互的基础。

在进行分类之前，首先给出有关线索和处理器的一般规则。现在，处理器是执行线索的实体，或事件发生的设备：

1. 由于线索要执行，因此有严格的正的持续时间。我们通常说线索的执行时间，但是还可能对线索发生（执行）的时刻感兴趣。行动是线索的退化形式，因此也有持续时间。

2. 在单处理器中，两个线索不能同时执行。这模拟一种物理基本概念：不会有两个物体同时占用同一空间。有时线索看起来好像是同时发生的，例如单处理器的分时，但是事实上，分时线索是交替发生的。即使线索不能在单处理器上同时执行，但是事件可以同时发生。（对于测试人员来说这确实是个问题。）

3. 事件有严格的正持续时间。如果把事件看做是在端口设备上执行的行动，则本规则就会变成规则1。

4. 两个（或多个）输入事件可以同时发生，但是一个事件不能在两个（或多个）处理器上同时发生。如果把端口设备看做是单独的处理器，则这个问题一下就清楚了。

5. 在单处理器中，两个输出事件不能同时开始，这是由线索执行引起的输出事件的直接后果。为了全面解释这个基本规则，我们既需要瞬时观点，也需要持续时间观点。假设两个输出事件中的一个的持续时间比另一个长得多，持续时间可能重叠（由于它们发生在单独的设备上），但是开始时间不会一样，如图15-1所示。举一个SATM系统中的例子，线索导致显示屏幕15，然后退出ATM卡。当ATM卡退出时间发生时，屏幕15仍然在显示。（这可能是一种细致的区分，我们还可以说端口设备是单独的处理器，端口输出事件实际上是处理器间的一种通信形式。）

6. 线索不能跨多个处理器。这种约定有助于定义线索。把线索定义在一个处理器上，就可以创建线索的自然端点，此外还导致多个单线索，而不是较少的复杂线索。在多处理器环境中，这种选择还会产生另外一种静止——跨处理器静止。

综上所述，这六个基本准则将所谓“健全行为”建立在第15.2节所定义的分类中的交互上。

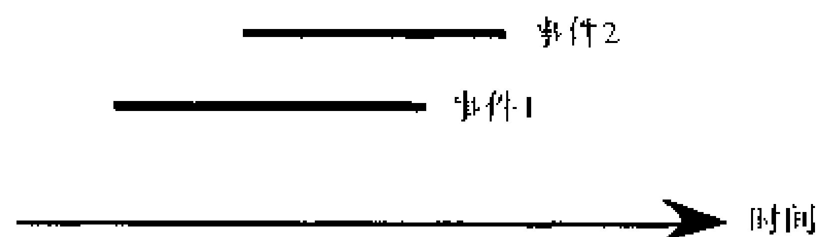


图15-1 重叠的事件

15.2 交互的分类

定位的两个方面，即时间和位置，构成交互分类的有用的切入点。有一些交互完全独立于时间，例如两个数据项的交互不考虑时间的因素。也有一些与时间有关的交互，例如一个事物是另一个事物的前提。我们可以通过区分单处理器和多处理器来细化静态/动态二分法。这两种考虑会产生一种有四个基本交互类型的二维平面，如图15-2所示：

单处理器中的静态交互。
多处理器中的静态交互。
单处理器中的动态交互。
多处理器中的动态交互。

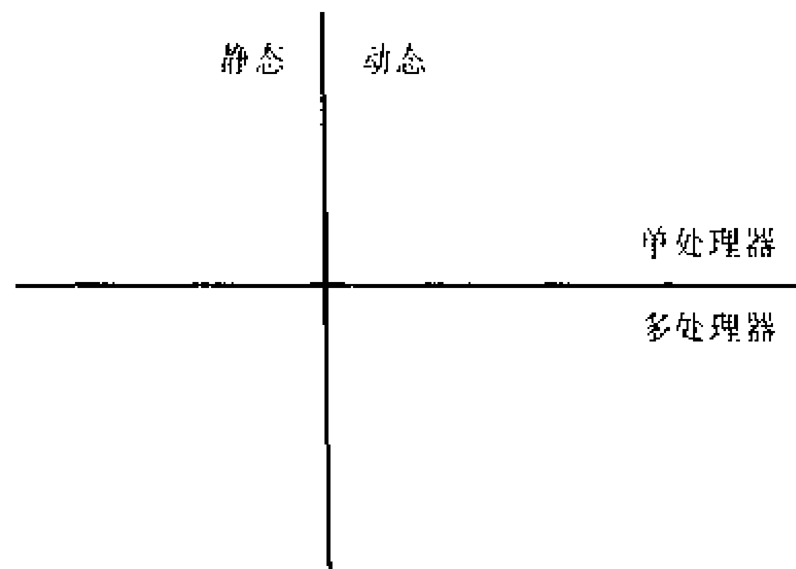


图15-2 交互类型

下面使用持续时间的概念细化这四个基本类型。线索和事件持续一定时间（因为要执行），因此不能是静态的。另一方面，数据是静态的，但是需要特别注意。考虑两个例子，对应于边的三元组（5，5，5）三角形的类型和支票账户余额。三角形的类型永远是等边三角形，时间永远不会改变几何形状，但是银行账户余额很可能会随时间变化。如果发生变化，则这些变化是由于这些线索的执行造成的，而这将是关键考虑

15.2.1 单处理器中的静态交互

在五种基本构造中，只有两种没有持续时间——端口和数据。端口是物理设备，因此可以将端口看做单独的处理器，并因而可简化问题的讨论。端口设备以物理方式交互，例如空间和能量消费，但是这对测试人员来说通常不重要。数据项以逻辑方式交互（与物理方式相对），而逻辑方式对测试人员很重要。我们常常以非形式化的方式讨论数据损坏和数据库完整性问题。有时会更精确一点，讨论数据的不兼容，甚至不一致问题。如果从亚里士多德那里借用一些术语，则可以非常具体地描述。（我们最终有机会使用第3章讨论过的谓词逻辑。）在下面的定义中，设p和q是关于数据项的谓词。例如，可以令p和q为：

p：账户余额 = 10.00美元

q：销售额 < 1800.00美元

定义

谓词p和q有：

如果两者不能都是真，则两者相反。

如果两者不能都是假，则两者次相反。

如果只有一个是真，则两者矛盾。

如果p真能够保证q为真，则q是p的次替代。

这些关系被逻辑学家叫做“相反正方形”，如图15-3所示，其中p、q、r和s都是谓词。

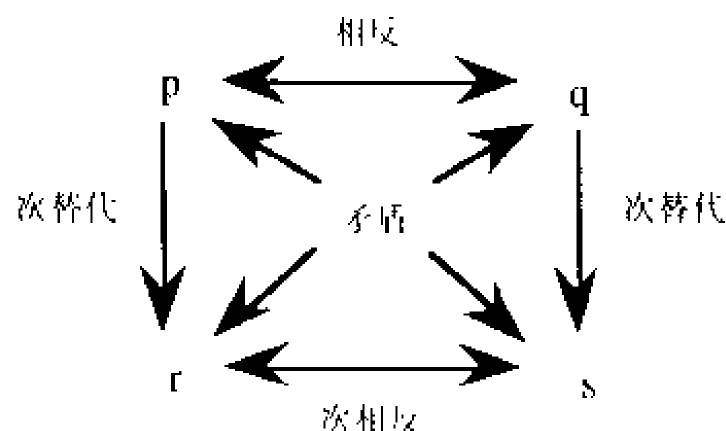


图15-3 相反正方形

亚里士多德逻辑在软件测试人员看来很神秘，但是有时数据交互正好符合相反正方形的特征：

- 当线索的前提由数据谓词构成时，相反或矛盾数据值可防止线索执行。
- 与语境有关的端口输入事件通常涉及矛盾的（或至少相反的）数据。
- 分支语句子句是矛盾的。
- 决策表中的规则是矛盾的。

单处理器中的静态交互正好可以与组合电路类比，也可以采用决策树和未屏蔽的事件驱动Petri网（EDPN）很好地描述。电话系统功能是交互的很好例子（Zave，1993）。一个例子

是呼叫方标识服务和未列出的目录号码之间的逻辑冲突。对于呼叫方标识，电话呼叫源的目录号码要由被叫方提供。如果没有列出目录号码的一方呼叫有呼叫方标识的一方时，就会出现冲突。呼叫方希望保守私密信息，而被叫方要知道谁在打入电话，究竟要满足谁呢？这两种功能是相反的，不能同时满足，但是可以都放弃。呼叫等待服务和数据线保证，是另一个相反功能的例子。如果某个公司购买了特别约定保证的数据线，则在这条线上的呼叫会经常被用来传输经格式化的二进制数据。如果这种数据线也有呼叫等待服务，则如果呼叫是对正在使用的线路发出的，则呼叫等待音会加在已经存在的连接上。如果该连接正在传输数据，则数据传输会被呼叫等待音破坏。这种情况的解决方案比较简单，即在进行数据传输呼叫之前，客户禁止呼叫等待服务。

15.2.2 多处理器中的静态交互

数据定位有助于解决电话系统例子中的矛盾。我们可期望呼叫等待数据和数据线保证都在同一个处理器中，因为两者都引用同一条订户线。因此，控制这条线的软件可以检查矛盾的线路数据。但是这对于呼叫方标识问题来说不是合理的期望。假设呼叫方所在电信局远离为呼叫方标识提供服务的电信局。由于这些数据存储在单独的位置（处理器）上，两者都不知道彼此的情况，因此只有当两者通过线索连接起来后，才有可能检测到这种相反性质。为了达到很精确的程度，我们可以说，相反关系作为一种跨多处理器的静态交互存在，当在两个电话局（处理器）中的执行线索交互时，就会出现失效。

呼叫转移是静态分布式交互的一个更好例子。假设在不同城市中有三个电话订户：

- 订户A位于密歇根的Grand Rapids
- 订户B位于亚利桑那的Phoenix
- 订户C位于马里兰的Baltimore。

进一步假设每个订户都有呼叫转移服务，且呼叫转移按以下方式进行：对A的呼叫转移到B，对B的呼叫转移到C，对C的呼叫转移到A。

这种呼叫转移数据是相反的，因为不能都为真。呼叫转移数据是局部于提供这种服务的电话局的，当订户定义新的转移目的地时设置。这意味着，电话局都不知道其他电话局中的呼叫转移数据，因此有分布矛盾。这是一种缺陷，但是只有当有人（除A、B或C之外）对这个呼叫转移圈呼叫时，这个缺陷才会变成失效。这种呼叫，比如说对订户B的呼叫，会在B的本地电话局中产生一个呼叫转移线索，导致对C的目录号码的呼叫。这又会在C的电话局中产生另一个呼叫转移线索，等等。现在请注意连接线索的存在，使我们从静态象限移出，进入动态交互象限中。潜在失效依然存在，只是位于分类的一个不同部分。

底线是静态连接本质上相同，不管是集中到单处理器中，还是分布在多个处理器上（但是很难检测什么时候分布）。另一种常见静态交互形式发生在数据库中的弱关系和函数依赖性上（集中或分布）。这些分布都是次替代形式。

15.2.3　单处理器中的动态交互

转移到动态象限意味着要考虑交互时间。除了其他问题之外，这还意味着必须从纯数据的交互扩展到数据、事件和线索之间的交互。我们还必须从相反正方形中严格说明性关系，转移到更加命令性的观点。有向图中的n-连接概念（请参阅第4章）正好适用于这种情况。图15-4给出了有向图中的四种n-连接形式。

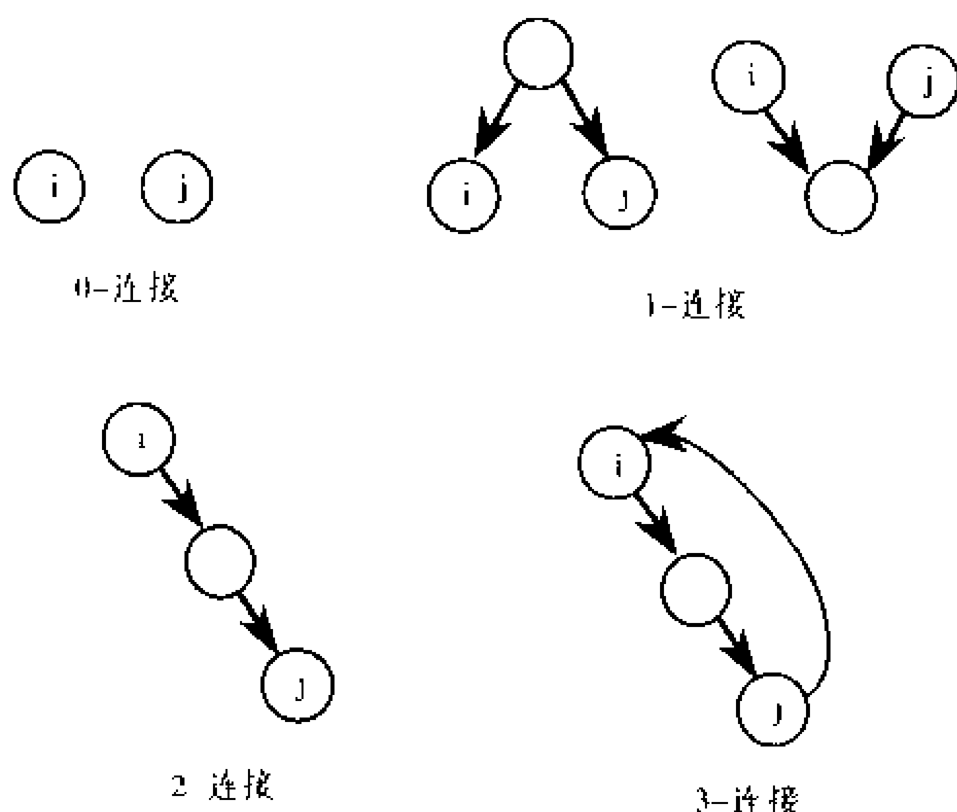


图15-4　n-连接的形式

即使数据之间的交互也具有n-连接形式。逻辑独立的数据是0-连接，次替代是2-连接。其他三种关系，即相反、矛盾和次相反，都适用于3-连接数据，因为这些都是双向关系。

有六种潜在概念对可以交互：数据与数据、数据与事件、数据与线索、事件与事件、事件与线索和线索与线索。每种概念对又可以进一步量化为n-连接四级，产生这个象限的24种基本类。以下讨论这些交互。请看以下四个例子：

1-连接数据与数据：当两个或多个数据项输入到相同的行动时发生。

2-连接数据与数据：当一个数据项用于计算（例如数据流测试）时发生。

3-连接数据与数据：当数据深度相关时，例如在重现和信号中发生。

1-连接数据与事件：与语境有关的端口输入事件。

我们不需要分析所有24种可能，因为只有当线索进行连接时交互缺陷才会变成失效。缺陷是潜在的，当线索进行连接时，潜在缺陷就会变成失效。线索只能以两种方式交互，通过事件或通过数据。在给出另一个定义之后，通过EDPN会对这一点有更清晰的认识。

定义

在EDPN中，外部输入是内度 = 0的地点，外部输出是外度 = 0的地点。

在图15-5所示的EDPN中，p1和p2是仅有的外部输入，p5、p9和p10是仅有的外部输出。尽管图中没有显示，不过作为前提和后果的数据地点，分别是外部输入和输出。对这一点的最好解释，就是外部输入的内度和外部输出的外度总是0。

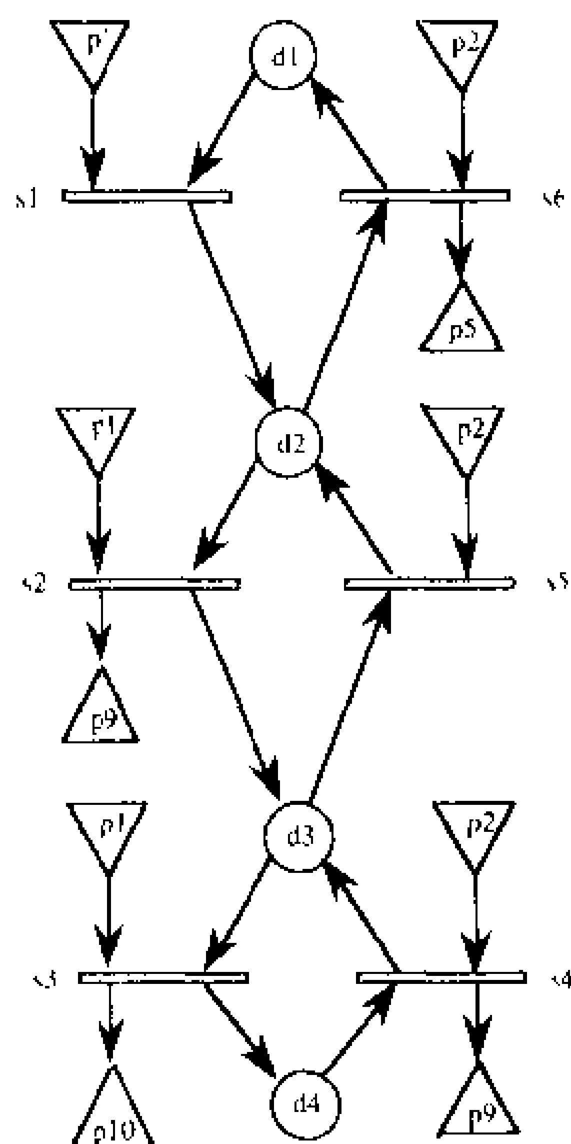


图15-5 在EDPN中的输入和输出

现在讨论问题的关键：可以通过线索的EDPN合成表示线索之间的交互。我们可以这样做：每个线索都有其自己的（惟一）EDPN，在每个EDPN中，地点和转移都有符号名称。一方面，这些名称是局部于线索的，但是从更高的角度看（当线索被组合时），本地名称必须解析为全局同义。例如，假设在一个SATM线索中，按下取消键被命名为端口输出事件p2，并且在另一个线索中，同样的事件叫做p6。当组合这两个线索时，同义必须合并为单一的名字。（本书前面已经指出，最好为端口事件起物理名称，而不是逻辑名称，同义解析也是提出这种建议的理由。）一旦同义被解析，个体线索就可以画成EDPN。由于线索只根据其外部输入和输出交互，因此下一步要标识每个线索的外部输入和输出集合。这些集合的交包含事件和被合成线索可以交互的地点。图15-6和15-7给出了这个过程。

线索1是序列<s1，s2>，线索2是序列<s3>。这些线索的外部输入是集合$EI1 = \{p1, d1\}$和集合$EI2 = \{p1, d2\}$，这些线索的外部输出是集合$EO1 = \{p3, d4\}$和集合$EO2 = \{p3, d4\}$。对这些集合做交运算，得到集合$EI = EI1 \cap EI2 = \{p1\}$和$EO = EO1 \cap EO2 = \{p3, d4\}$。集合EI和EO包含线索1和2可能交互的外部输入和输出。请注意，合成线索的外部输入和输出是$EI1 \cup$

EI2和EO1∪EO2。外部事件将永远是外部的，但是外部数据可能不是外部的（呼叫转移循环中就出现这种情况。）

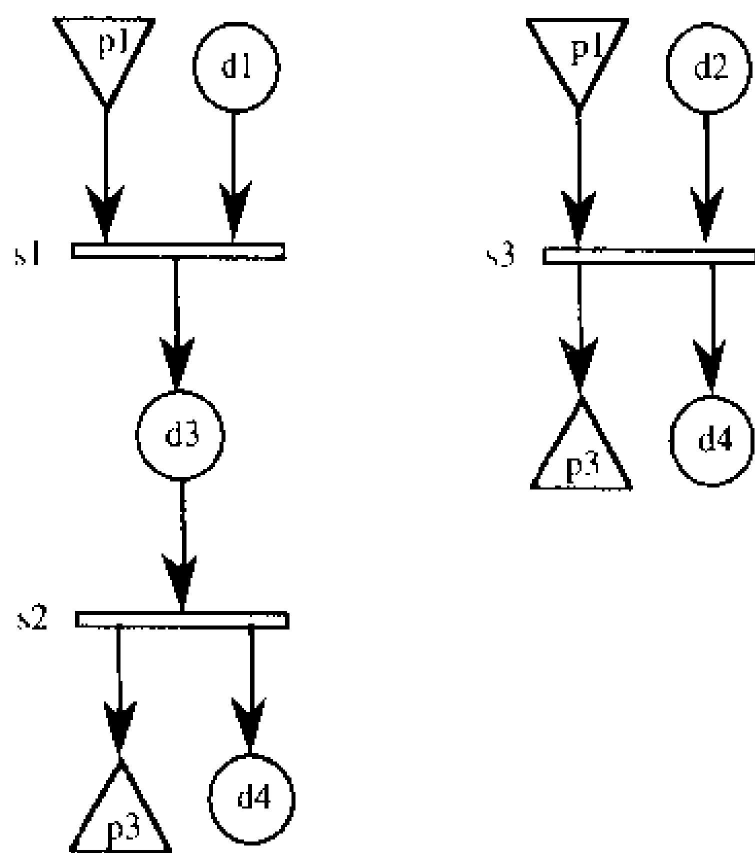


图15-6 两个EDPN线索

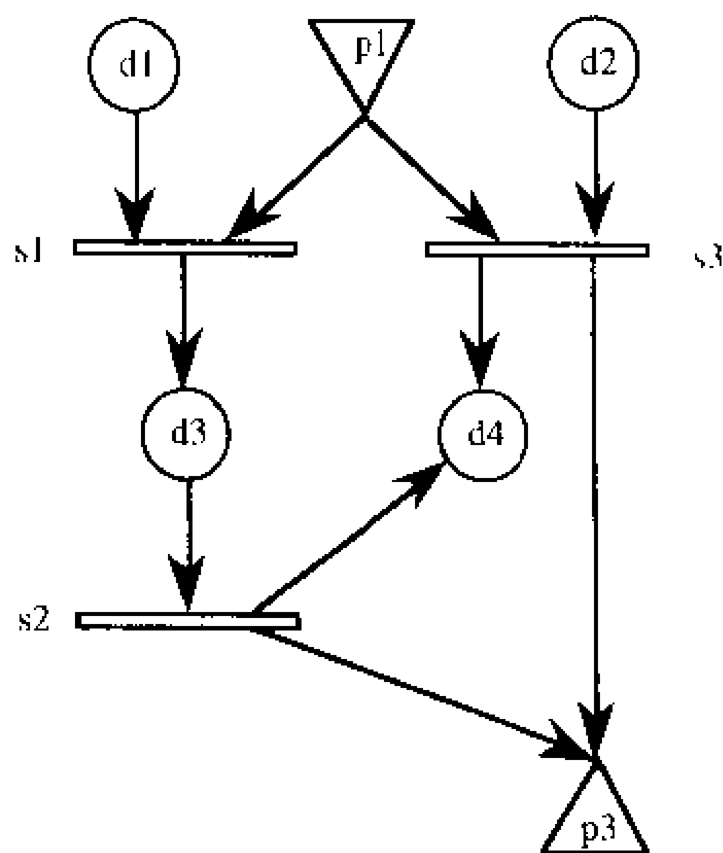


图15-7 EDPN线索的合成

可以把这个过程形式化为另一个定义。设T1和T2是两个EDPN线索，其中的同义地点已经得到处理，具有外部输入和输出集合EI1、EI2、EO1和EO2。此外，设T是线索T1和T2的合成，其中EI = EI1∩EI2和EO = EO1∩EO2是合成线索T的外部输入和输出集合

定义

线索T1和T2是：

0-连接，如果$EI1 \cap EI2 = \varnothing$，$EO1 \cap EO2 = \varnothing$，$EI1 \cap EO2 = \varnothing$，$EO1 \cap EI2 = \varnothing$。

1-连接，如果有$EI \neq \varnothing$或$EO \neq \varnothing$。

2-连接，如果有$EI1 \cap EO2 \neq \varnothing$或$EI2 \cap EO1 \neq \varnothing$。

3-连接，如果有$EI1 \cap EO2 \neq \varnothing$并且$EI2 \cap EO1 \neq \varnothing$。

将上面的定义与第4章的n-连接定义进行比较。第4章关心的是节点对偶之间的关系，本章关心的是线索之间的关系。通过构造相当精细的有向图，其中的节点表示删除了外部输入和输出的线索，边表示根据被删除的外部输入和输出地点对线索的连接，可以消除这些有一定重叠的定义。图15-8给出了图15-7的一种合成图。请注意可以怎样直接通过输入地点p1和输出地点p3和p4看出线索是1-连接的。

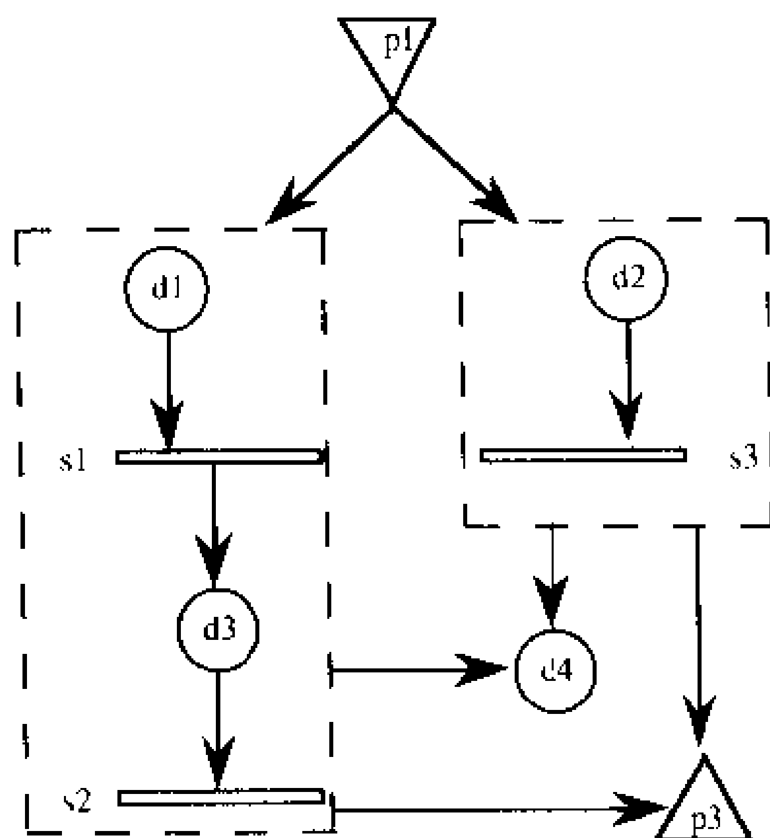


图15-8 n-连接线索

n-连接定义确切地描述了线索在动态、单处理器象限中交互的方式。只遗漏了一件事：与一般有向图中的n-连接一样，通过线索链，而不是这里介绍的相邻线索，可以连通。（想像12个州内的12个电话订户构成的呼叫转移循环。）根据我个人的测试经验，现场问题报告中描述的很多问题，都是未预料的1-、2-或3-连接的线索实例，而这些线索被误认为是0-连接的。

需要对线索之间1-连接的两种形式做特殊说明。如果两个线索通过一个公共输入（事件或数据）连接，那么该交互就是经典形式的Petri网冲突。（在图15-7中，线索1和2是通过端口输入事件p1连接的1-连接，这也是与语境有关的端口输入事件的一个例子。）如果考虑一种标记，其中每个线索的第一个转移都被允许，则触发其中一个线索会消除来自公共输入地点的记号，并禁止其他线索触发。如果两个线索是通过一个输入地点（同样也可以是事件或数

据）连接的1-连接，则不会有冲突发生，但是有一种需要注意的容易被误解的地方。当一个线索执行时，公共输出地点会被标记，但是通常不知道是哪个线索。在SATM系统中，当显示屏幕10时，就会出现这种情况（暂时不能处理取款）。屏幕10可以显示，因为ATM已经没有现金，或取款通道有问题。这种输出引起的非明确形式在现场问题报告中经常出现。很多时候是因为线索之间未预料到的交互，有些线索可能在其他之前很长时间就已经执行了。

端口事件不能既是输入又是输出，因此线索为2-连接的惟一方式是通过数据地点；而且由于3-连接是双向的2-连接，因此线索为3-连接的惟一方式是通过数据地点。这在一定程度上可简化交互，因为可能出现的数量降低。通过数据交互的实际问题是有可能涉及很长的持续时间。这里我们可以稍加改进：即区分只读数据和读写数据。只读数据永远不会改变，因此可以说有无限（或不定的）持续时间。读写数据在写入时显然会改变，因此在连续写入之间有持续时间。现实问题是，很少被使用（写）的数据可能是引起两个线索成为2-或3-连接的连接，第一和第二个线索之间的长时间分离，会很难诊断，很难通过测试用例反映出来。

在SATM系统中有一些动态交互。首先请注意，我们处理的是一台ATM终端，因此是通过单处理器中的动态交互连接。从技术上看，使用的是两个处理器，因为SATM系统要从中央银行获得账户信息，但是可以把中央银行当做端口设备来处理交互。在SATM系统中有多个与语境有关的端口输入事件，因为第14章已经详细讨论了这些端口输入事件，因此下面讨论线索之间的1-和2-连接交互。

很容易实现向给定银行账户存款或取款的线索。这些都是通过账户余额数据地点的1-和2-连接。试图提取超过或低于现有余额现金的取款线索，对余额地点是1-连接。成功的取款（所请求的金额低于余额）显然会改变余额，因此余额地点前后实例是通过取款线索的2-连接。如果添加一个存款线索，则可以得到更丰富的交互，并且所有问题的关键，是交互的执行时间顺序。

在以上讨论中存在的另一个微妙问题与我们如何描述这种交互线索有关。一种可能是依赖特定值，都针对线索输入部分中的前提和所键入的金额。为了明确，可以作为前提假设余额为50.00美元，定义线索T1为取款40.00美元，定义线索T2为取款60.00美元。很明显，线索T2不会执行，线索T1将执行。为了更精确，我们定义取款金额的两端口输入：p1取款金额 = 40.00美元，p2取款金额 = 60.00美元。在这种情况下，线索T1将包含显示屏幕11的端口事件（请取走现金），而T2将包含显示屏幕8的端口事件（余额不足）。这样就会出现上面提到的微妙问题。为了更精确地定义，我们还清除了这个冲突，因为事先已经定义了输出要出现的端口。什么时候这些值存在，什么时候描述（说明）线索，或什么时候执行这些线索？测试人员经常使用显式方法，但是由于先天不足，不太适合需求规格说明。我们也可以更宏观地说线索T1是金额小于余额的取款，线索T2是金额大于余额的取款，并且直到执行时才确定余额。（如果采用可执行规格说明作为快速原型，则这是一种好的选择。）后一种选择的问题是，测试人员不能提供测试用例的预期输出部分。微妙之处在于什么时候线

索的路径能够确定：在线索开始执行之前确定，例如在测试用例中确定，或是在执行线索时确定，例如在快速原型中确定。第15.3节在讨论交互、合成和确定性之间的关系时，还会讨论这个问题。目前测试人员可能喜欢能够自己控制线索结果的思路，但是需求定义人员则不喜欢。

15.2.4 多处理器中的动态交互

多处理器之间的动态交互在我们的分类中是最复杂的。由于线索和事件可以同时执行，因此，严格串行、确定的行为要由并发行为取代。按Robin Milner的话说，“并发性造成非确定性”（Milner，1993）。额外的复杂性在每个象限所要求的模型中也可以看到：决策表能够满足静态交互的要求，有限状态机表示多处理器上的动态交互。多处理器上的动态交互需要通信有限状态机或某种形式Petri网的表达能力。我们将在土星牌挡风玻璃雨刷系统中看到这些交互。

土星汽车上的挡风玻璃雨刷由一个控制杆和一个刻度盘控制。控制杆有四个位置，即“关”、“间歇”、“低速”和“高速”；刻度盘有三个位置，分别用数字1、2和3表示。刻度盘位置表示三种间歇速度，只有当控制杆位于“间歇”时，刻度盘位置才有意义。以下决策表显示出控制杆和刻度盘位置对应的挡风玻璃雨刷速度（每分钟摇摆次数）

控制杆位置	关	间歇	间歇	间歇	低速	高速
刻度盘	-	1	2	3		- -
雨刷速度	0	4	6	12	30	60

如果仔细考虑土星牌挡风玻璃雨刷系统，就会发现三种自然交互设备：控制杆、刻度盘和雨刷。控制杆有两种事件：可以上移一个位置或下移一个位置。类似地，刻度盘也有两个事件：可以顺时针转动或逆时针转动。可以把这些事件分解到更细的粒度，如表15-1所示。

表15-1 土星牌挡风玻璃雨刷中的端口输入事件

端口输入事件	描　述
ie1	控制杆从“关”到“间歇”
ie2	控制杆从“间歇”到“低速”
ie3	控制杆从“低速”到“高速”
ie4	控制杆从“高速”到“低速”
ie5	控制杆从“低速”到“间歇”
ie6	控制杆从“间歇”到“关”
ie7	刻度盘从1到2
ie8	刻度盘从2到3
ie9	刻度盘从3到2
ie10	刻度盘从2到1

雨刷设备产生六种输出事件，即六种不同雨刷速度（以每分钟摇摆次数表示），如表15-2所示

表15-2　土星牌挡风玻璃雨刷中的端口输出事件

端口输出事件	描　述
oe1	每分钟0次
oe2	每分钟4次
oe3	每分钟6次
oe4	每分钟12次
oe5	每分钟30次
oe6	每分钟60次

控制杆和刻度盘有限状态机如图15-9所示。请注意，我们可以很容易地显示导致状态转移的事件，但是有些相关输出（由问号表示）是为确定的。例如，当把控制杆从“关”推向“间歇”后，我们不能断定具体的端口输出事件，因为我们不知道刻度盘机器的状态，我们也不能断定刻度盘机器的输出状态，因为我们不知道控制杆是否处于“间歇”位置，控制杆和刻度盘并发驱动雨刷。

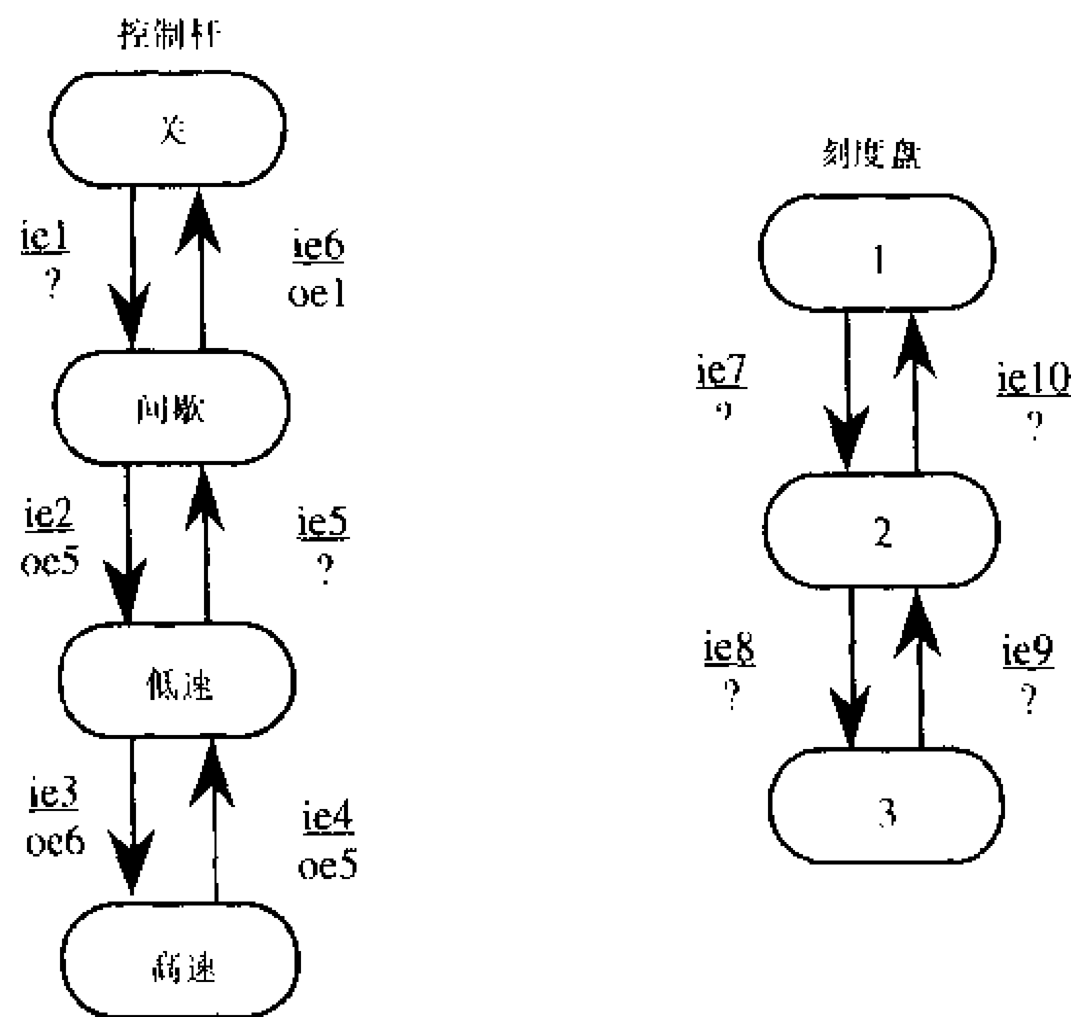


图15-9　控制杆和刻度盘有限状态机

我们使用EDPN合成控制杆和刻度盘有限状态机，并且这种合成会确切地描述交互。图15-5中的EDPN等效于控制杆有限状态机，刻度盘有限状态机的EDPN如图15-10所示。图15-11所使用的EDPN转移、地点和事件，被归纳在表15-3中

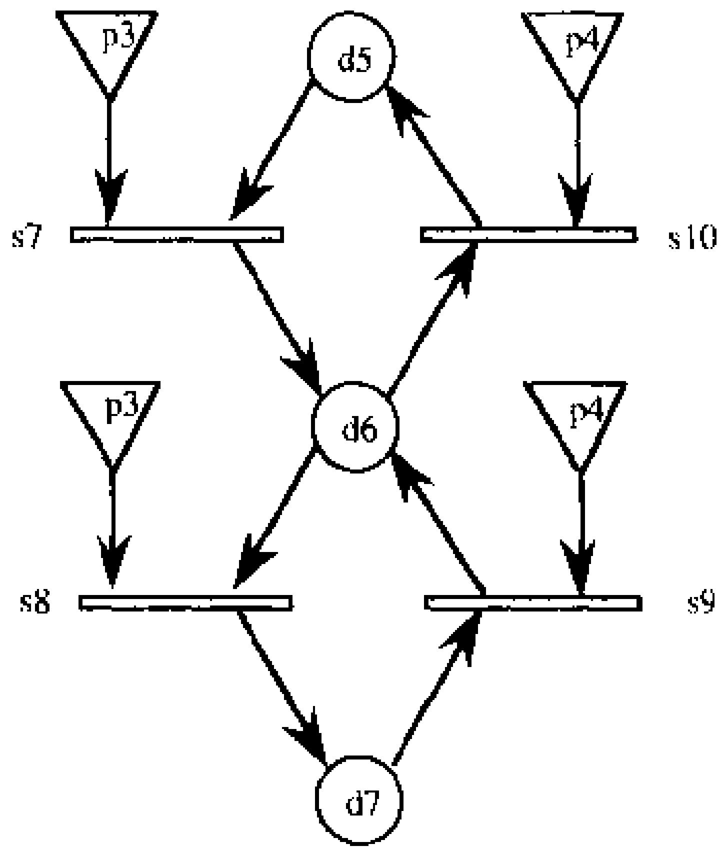


图15-10　刻度盘EDPN

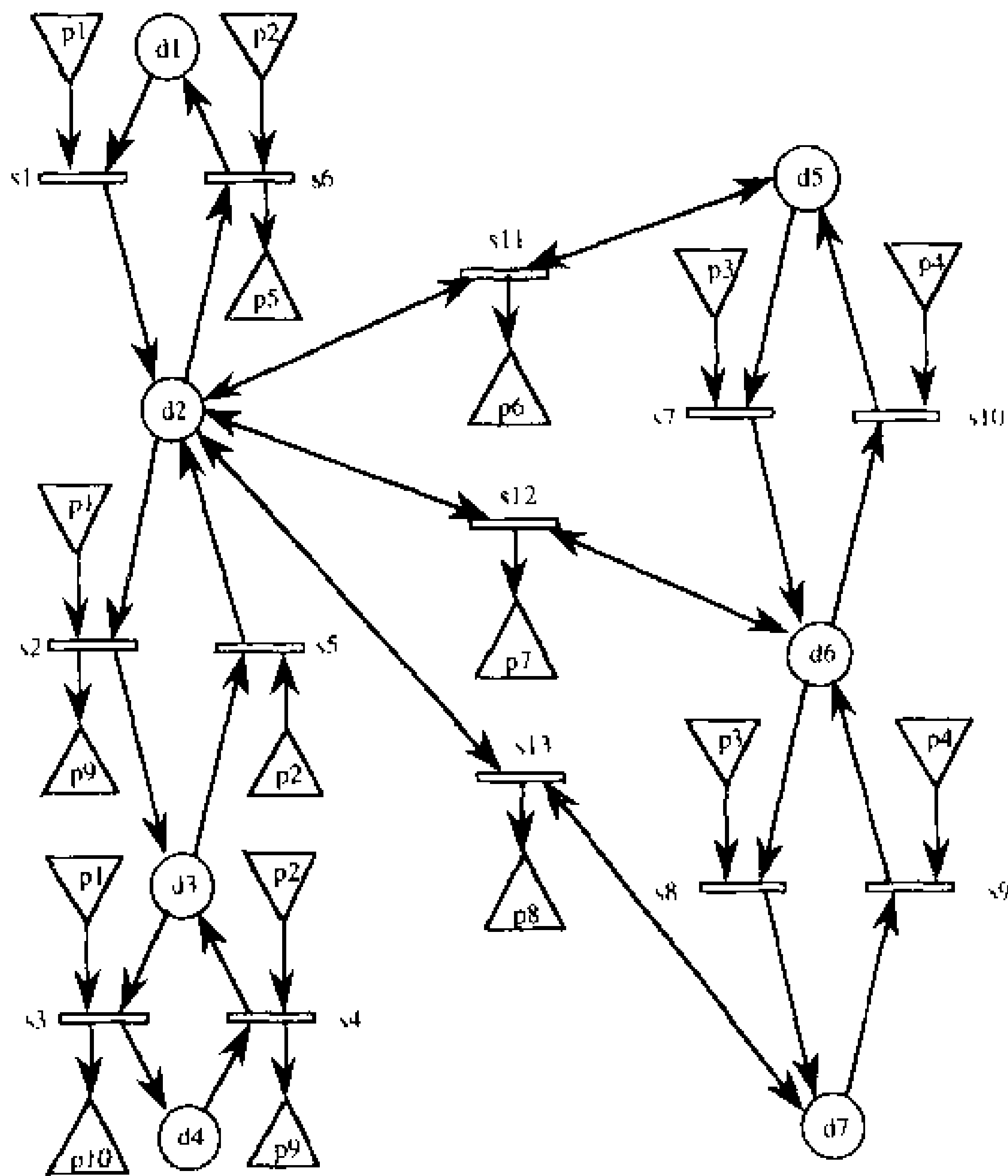


图15-11　挡风玻璃雨刷系统EDPN

表15-3

用户输入	预期输出
控制杆上移一个位置	每分钟6次摆动
控制杆上移一个位置	每分钟30次摆动
刻度盘上移一个位置	每分钟30次摆动
控制杆下移一个位置	每分钟12次摆动
控制杆下移一个位置	每分钟0次摆动

为了把有限状态机构造为EDPN，状态要变成数据地点，状态转移变成EDPN转移。引起状态转移的事件成为端口输入事件，与状态转移关联的输出变成端口输出事件。在控制杆EDPN（如图15-5所示）中，向“关”、“低速”和“高速”状态转移，确定相关的端口输出事件不要求与刻度盘EDPN交互（p5、p9和p10）。此外，到“间歇”状态的转移（s1和s5）没有端口输出事件。在刻度盘EDPN（如图15-10所示）中，三个刻度盘位置是三个数据地点，刻度盘事件（p3和p4）是引起转移的输入。刻度盘EDPN没有端口输出事件，因为在刻度盘EDPN中，刻度盘事件永远不是转移的输出。

控制杆和刻度盘EDPN之间的交互如图15-11所示（请参阅表15-4）。图中给出了地点d2、d5、d6和d7，转移s11、s12和s13，端口事件p6、p7和p8，分别是“间歇”状态、刻度盘位置状态和三种间歇雨刷速度。各端带箭头的边（例如d2和s11之间的边）表示地点是相关转移的输入和输出。这看起来有点人为规定的味道，但是却可以使标记工作顺利进行。

表15-4 EDPN挡风玻璃雨刷元素

元素	描　述
s1	“关”到“间歇”的转移
s2	“间歇”到“低速”的转移
s3	“低速”到“高速”的转移
s4	“高速”到“低速”的转移
s5	“低速”到“间歇”的转移
s6	“间歇”到“关”的转移
s7	1到2的转移
s8	2到3的转移
s9	3到2的转移
s10	2到1的转移
s11	提供每分钟6次摆动
s12	提供每分钟12次摆动
s13	提供每分钟20次摆动
d1	控制杆位于“关”
d2	控制杆位于“间歇”
d3	控制杆位于“低速”
d4	控制杆位于“高速”
d5	刻度盘位于1
d6	刻度盘位于2
d7	刻度盘位于3

（续）

元素	描述
p1	控制杆上移一个位置
p2	控制杆下移一个位置
p3	刻度盘上移一个位置
p4	刻度盘下移一个位置
p5	提供每分钟0次摆动
p6	提供每分钟6次摆动
p7	提供每分钟12次摆动
p8	提供每分钟20次摆动
p9	提供每分钟30次摆动
p10	提供每分钟60次摆动

我们可以执行图15-11中的EDPN，但是需要设计（也就是有理化）初始标记。请注意，没有地点是外部输入。这意味着必须任意地确定刻度盘和控制杆的起始位置。实际情况是，两个设备在一个时刻都只会在一个位置上（一种相反的形式），因此，d1、d2、d3和d4中的一个总是会被标记。d5、d6和d7也有类似的情况。我们假设汽车发货时控制杆在关的位置，刻度盘在位置1。表15-5给出了以下场景的标记顺序：

图15-11中完整土星牌挡风玻璃雨刷系统EDPN显示的多处理器之间动态交互的复杂性还仅仅是开始。首先，考虑三方面的复杂性，即控制杆EDPN的3、刻度盘EDPN的2，以及完整EDPN的20。出现这种复杂性与交互有直接关系。下面考虑由这个网络的各种由串行标记产生的复杂性，然后再考虑允许并发执行时会产生的复杂性。

如果仔细研究图15-11中EDPN的表15-5中的标记顺序，就会清楚这一点。首先请注意会出现多点事件静止，例如标记步骤m0和m2。下面解释一下步骤m3和m5的网络。首先要澄清端口输出事件的性质（雨刷速度）。当发生端口输出时，其持续时间有多长？是否继续保持摆动速度，还是必须重复执行？这些风格都在图15-11中表示出来。在步骤m3，当转移s11触发时，会考虑其输入地点（d2和d5），并执行p6（每分钟6次摆动）。由于开放其他转移，因此可以认为s11在不断触发，直到其他事件发生。这是一种很奇特的事件静止形式，但显然是系统的一个稳定状态。其他形式的事件静止通过标记步骤m5表示，其中s2被触发，产生端口输出p9（每分钟30次摆动）。现在我们认为p9的持续时间与d3被标记的时间一样长。在p9运行时，用户可以引发刻度盘端口事件（步骤6和7），但对雨刷速度没有影响。可能的标记使我们加深这些交互复杂性的理解。

时间因素进一步增加了复杂性。考虑端口输入事件之间的时间区间，以及我们所说的系统响应时间。最高的雨刷摆动速度是每分钟60次，驾驶员可以轻易地在1秒钟之内，将控制杆从“关”位置推到“高速”位置。对于中间的端口事件（p9以及p6、p7和p8中的一个）会出现什么情况呢？在这种细节上这个模型是不完整的。标记使我们能够处理并发事件：如果两个事件（或转移）恰好在同一个时间点上发生，可以很容易地通过在标记向量中标记其相应位置表示。还可以修正触发规则，以允许转移被同时触发，只要转移发生在不同的处理器中。可以采用类似用来描述电路时序图的图形表达所有这些。（这些图形还类似乐谱，在乐谱中每个音部都有自己的记号。音乐还有节拍，是相等持续时间的度量。）表15-6和15-7给出了表示表15-5标记并发行为的两种可能的方式。

表15-5 样本场景的标记

m	d1	d2	d3	d4	d5	d6	d7	p1	p2	P3	p4	p5	p6	p7	p8	p9	p10	描 述
0	1				1													关，1
1	1				1			1										控制杆上移，s1开放
2		1			1													s1触发，s11开放
3		1			1								1					s11触发、s11开放
4		1			1			1										控制杆上移、s2和s11开放
5			1		1											1		s2触发
6			1		1					1								刻度盘上移，s7开放
7			1			1												s7触发
8			1			1			1									控制杆下移，s5开放
9		1				1												s5触发，s12开放
10		1				1								1				s12触发
11		1				1			1									控制杆下移，s6开放
12	1					1						1						s6触发

表15-6 样本场景的并发行为

时间	0	1	2	3	4	5	6	7	8	9	10	11	12
控制杆													
d1	1	1											
d2				1	1								1
d3							1	1	1	1	1		
d4													
s1			1										
s2						1							
s3													
s4													
s5													
s6												1	
p1		1			1								
p2											1		
刻度盘													
d5	1	1	1	1	1	1	1	1					
d6										1	1	1	1
d7													
s7									1				
s8													
s9													
s10													
p3								1					
p4													
雨刷													
s11													
s12													
s13													
p5													
p6				1	1	1							
p7													1
p8													
p9							1	1	1	1	1	1	
p10													
时间	0	1	2	3	4	5	6	7	8	9	10	11	12

表15-7　表15-6的压缩版本

	时间	0	1	2	3	4	5	6	7	8	9	10	11	12
控制杆	数据	d1	d1		d2	d2		d3	d3	d3	d3	d3		d2
	转移			s1			s2						s5	
	输入		p1			p1						p2		
	输出							p9	p9	p9	p9	p9	p9	
刻度盘	数据	d5	d5	d5	d5	d5	d5	d5	d5		d6	d6	d6	d6
	转移									s7				
	输入								p3					
	输出													
雨刷	数据													
	转移													
	输入													
	输出				p6	p6	p6							p7

15.3　交互、合成与确定性

非确定性问题具有科学和哲学深层问题的背景。爱因斯坦不相信非确定性，他有一次说，他很怀疑上帝是否正在和宇宙玩骰子。非确定性一般指随机事件的结果，实际上，如果存在真正的随机事件（输入），我们能够预测其结果吗？这种争论的逻辑终点可归结到哲学/神学的自主意志与宿命论问题上。对于测试人员，幸运的是非确定性的软件版不那么严重。读者可以把本节看做是技术方面的专题文章。本节内容的基础是我的经验和使用EDPN框架的分析，我发现这些可以合理地回答非确定性问题，读者利用自己的经验也可以回答。

首先给出可行的确定性定义，以下是两种可能的定义：

1. 系统是确定的，如果给定输入我们总可以预测其输出。
2. 系统是确定的，如果对于给定输入集合，系统总是产生相同的输出。

第二种观点（可重复输出）与第一种观点（可预测的输出）相比，严厉程度较低，因此我们使用第二种作为这里的工作定义。这样，非确定性系统就是至少有一组输入会产生两种不同的输出集合。很容易构建非确定性有限状态机，图15-12给出了一个例子。

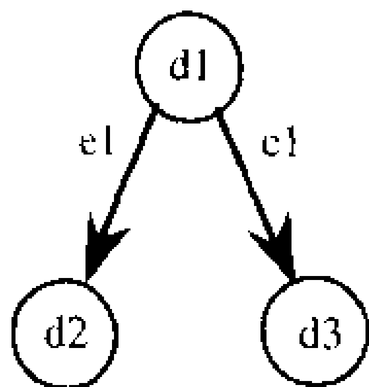


图15-12　非确定性有限状态机

当图15-12中的机器处于状态d1时，如果事件e1发生，会发生到状态d2或d3的转移。

构建非确定性有限状态机这样容易，前面为什么还要讨论那么多的确定性问题呢？第12章曾经努力地将系统的现实性从系统行为模型中剥离开来。有限状态机是现实性模型，只近似表示现实系统的行为。这也是为什么选择合适模型非常重要的一个原因，因为我们要使用最好的近似。粗略地说，决策表模型适合静态交互，有限状态机模型适合多处理器中的动态交互，某种形式的Petri网适合多处理器中的动态交互。在继续讨论之前，我们首先指出其他两种模型中的非确定性实例。在多命中决策表中，输入（条件桩中的变量）要根据多个规则选择。在Petri网中，当开放多个转移时会出现非确定性。选择哪个规则执行，或选择触发哪个转移，要由外部代理做出。（请注意，这种选择实际上是输入。）

这样，我们的非确定性问题就归结为EDPN中的线索，这也是交互、合成和确定性的汇集点。为了在讨论中加入实际成份，下面考虑以前使用过的SATM线索：

T1：取款40.00美元。

T2：取款60.00美元。

T3：存款30.00美元。

线索T1、T2和T3通过账户余额数据地点交互，并且可以在不同的处理器中执行。最初余额为50.00美元。

从线索T1开始：如果没有其他线索执行，则T1能够正确地执行，使余额变成10.00美元。假设首先执行线索T2，实际上应该称做“试图取款60.00美元”，因为如果不执行其他线索，则T2会导致显示资金不足屏幕。我们实际上应该把T2分解为两个线索，即T2.1，成功地完成取款，最后显示屏幕11（取走现金），以及T2.2，取款失败，最后显示屏幕8（资金不足）。现在通过线索T3增加一些交互。线索T2和T3是通过余额数据地点的2-连接。如果T3在T2读取余额数据之前执行，则T2.1发生，否则T2.2发生。从这里可以看出确定性两种观点之间的差别：当T2的EDPN开始执行时，我们不能预测结果是什么（T2.1还是T2.2），因此根据第一种定义，这是非确定性的。但是根据第二种定义，我们可以重新创建T2和T2之间的交互（包括时间）。如果这样做，就可以将行为作为合成EDPN标记捕获，可满足确定性的可重复定义。

对于确定性问题，我们有一种折衷解决方案，至少可以适用于采用EDPN表达的测试线索。我们还可以再向前走，说线索是局部非确定性的，因为线索的结果不能使用线索的局部信息预测。通过挡风玻璃雨刷系统可以看到这一点。如果我们局限于控制杆有限状态机，则当控制杆推到“间歇”位置时，就不能确定输出是什么，因为不知道刻度盘的位置。但是从全局观点看，就不存在非确定性问题了，因为所有输入都是已知的。这对测试人员意味着，当测试包含外部输入（特别是数据地点）的线索时，重要的是要测试与所有其他可以通过外部输入构成n-连接的线索的交互。

15.4　客户-服务器测试

客户-服务器（CS）系统很难测试，因为这类系统具有最困难形式的交互，即跨多处理器的动态交互。这里，我们可以利用强理论开发的优势。客户-服务器系统总是有至少两个处理器，在一个处理器上保存并执行服务器软件，在另一个（通常是多个）处理器上执行客户软件。主要成份一般是数据库管理系统、使用数据库的应用程序以及产生用户定义输出的表示程序。这些组件的位置决定胖服务器与胖客户差别（Lewis，1994）（如图15-13所示）。

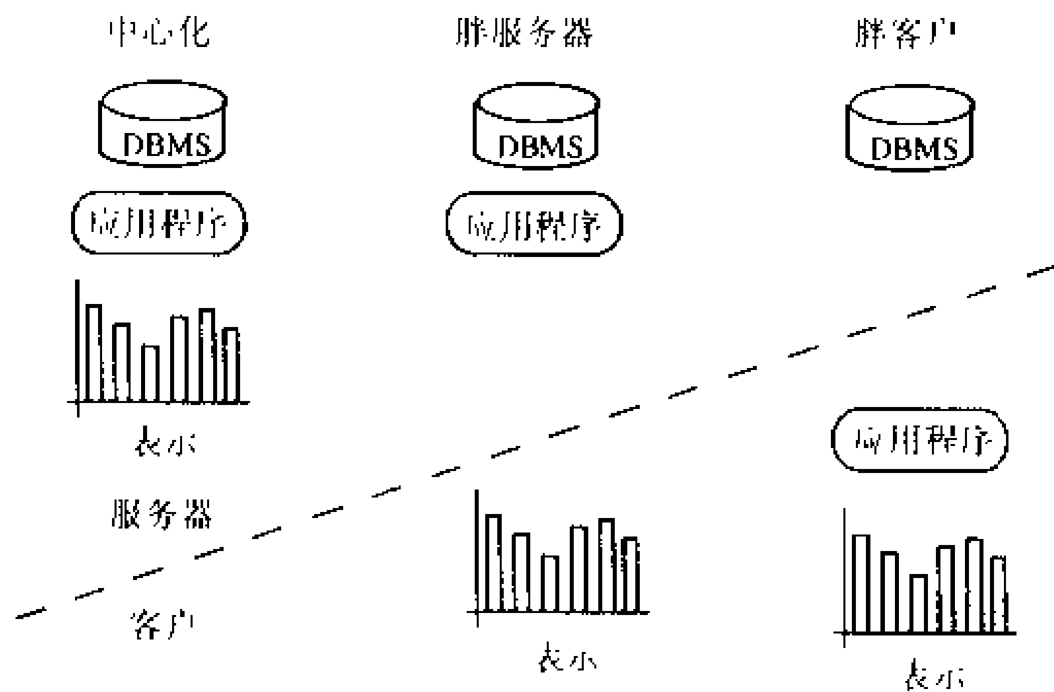


图15-13　胖客户和服务器

客户-服务器系统还包含将客户与服务器连接的网络、网络软件以及客户的图形用户界面（GUI）。更糟糕的是，我们可以通过也不相同的客户处理器区分同构和异构CS系统。SATM系统多终端版本是一种胖客户系统，因为中央银行只完成很少的事务处理工作。

在我们的EDPN线索表述中，把线索局限于一个处理器。对于CS系统，需要跨CS边界的线索序列定义，叫做CS事务。典型的CS事务从客户处理器发出请求（或查询）开始。该请求通过网络（可能通过定时方式）传递给服务器，服务器上的某个应用程序使用数据库管理系统（DBMS）对请求进行处理，结果通过网络传递给客户，在客户上将结果映射到用户定义的输出格式上。如果在CS事务处理过程中出现问题，则会出现各种相互指责情况：客户怪罪服务器，服务器怪罪客户，服务器和客户都怪罪网络。

CS事务DBMS部分中的线索都属于静态串行类型，并且由于要么是商业化产品，要么是非常稳定的应用程序，因此不需要测试。如果需要测试的话，几乎总是功能性测试，因为极少能够得到商业化产品的源代码。服务器应用程序中的线索也主要是静态串行交互，一般也是稳定的现有应用程序，可能比DBMS更容易出现错误（出现失效是更好的说法）。CS事务的网络软件部分也很可能是商业化应用程序，因此大多数CS事务测试主要在客户处理器上进行。（从测试观点看，）最有意义的是GUI部分。CS事务的用户部分一般构建在允许用户开发WIMP接口（窗口、图标、菜单和下拉式菜单）的商业化应用程序内部，而这里正是

出现乐趣的地方 客户可以以任意方式跨多个窗口，但是结果必须是兼容的，按照我们的术语说，就是可以把窗口看做是一种有限状态机，窗口之间的转移对应于相互通信的有限状态机。对窗口用户提供的所有行动都是端口输入事件，这些行动的结果是端口输出事件。因此，CS事务的客户部分是跨多处理器（窗口）的动态交互线索，我们已经知道这是最复杂的一种交互。

这种框架至少向CS测试人员提供一种支持测试策略的严格方法，可以通过覆盖指标度量，操作剖面的概念（请参阅第14章）是非常合适的，因为测试所有可能的交互会非常费时 这种框架的另一种优点是，一旦客户线索被满意地测试，下一步惟一要关心的只是与其他线索的交互，这应该是一个很重要的步骤，因为有单独客户的线索之间的大量潜在的n-连接，非常类似SATM场景中多个用户以相同账户存款和取款情况

15.5 参考文献

Lewis, T. and Evangelist, M., Fat servers vs. fat clients the transition from client–server to distributed computing. *American Programmer.* Vol 7 No. 11, November 1994, pp. 2–9

Milner, Robin, Elements of Interaction (1993 Turing Award Lecture), *Communications of the ACM*, Vol. 36, No. 1, January 1993

Zave, P., Feature interactions and formal specifications in telecommunications, *IEEE Computer,* August 1993.

15.6 练习

1. 图15-3清晰地给出了集合论的联系 设谓词p、q、r和s是关于集合S和P的以下集合论陈述：

p: $S \subseteq P$ （所有S都是P）

q: $S \cap P = \varnothing$ （没有S是P）

r: $S \cap P \neq \varnothing$ （有些S是P）

s: $S \not\subset P$ （有些S不是P）

请说明相反正方形中的关系适用这些集合论命题 （亚里士多德将自然语言分为演绎 ）

2. 请找出并讨论表15-4事件行中的事件n-连接例子

第五部分

面向对象的测试

第16章

面向对象的测试问题

20世纪90年代后半期，面向对象软件测试在理论和实践上都得到迅速发展 在面向对象程序设计、系统、语言和应用（OOPSLA）会议和厂商资助的测试会议上，有很多面向对象测试的论文发表，并提供很多讲座。一个Web站点就有18 000个面向对象测试论文的连接（www.cetus-links.org/oo_testing.html）。到21世纪开始时，人们越来越多地提出，“什么是面向对象单元？”“怎样最好地为面向对象应用程序建模？”面向对象软件最初的希望之一，是对象可以不加修改或额外测试地重用 这种希望所基于的假设是，计划周密的对象封装“属于一起的”功能和数据，并且一旦这种对象被开发和测试，就成为可重用的组件 最近人们对面向特征程序设计的研究（Kiczales，1997），是对面向对象范例某些局限性的一种回答，越来越多的人认识到，对面向对象软件的估计没有什么理由这样乐观，因为与传统软件相比，面向对象软件具有潜在更严重的测试问题，积极方面、新诞生的统一建模语言（UML）是面向对象技术多个方面的很强的统一（和推动）力量

本章的目标是找出由面向对象软件引出的测试问题。首先，我们有测试层次问题，而这又要求澄清如何划分面向对象单元。其次，要考虑合成策略（与功能分解相反）引出的一些问题。面向对象软件的特征，是继承、封装和多态性，因此本章将研究能够扩展传统测试，以解决这些问题的方式。第五部分中的其余各章将讨论面向对象软件的类测试、图形用户界面（GUI）测试、集成与系统测试、基于UML的测试以及数据流应用程序的测试，本章最后将给出两个说明这些问题的例子。

16.1 面向对象测试的单元

传统软件对“单元”有各种定义，其中适用于面向对象测试的两种定义是：

- 单元是可以编译和执行的最小软件组件。
- 单元是决不会指派给多个设计人员开发的软件组件。

这些定义可能存在矛盾。有些实际应用程序有很大的类，显然与一个设计人员一个类的定义冲突。在这种应用程序中，看起来最好把面向对象单元定义为很可能最终构成类操作子集的一个人的工作。在极端情况下，面向对象单元可以是只包含单一操作或方法所需属性的子类。（第五部分中的“操作”指类函数定义，“方法”指其实现）对于这种单元，面向对象的单元测试就归结为传统测试 这是一种很不错的简化，但是存在问题，因为它把大量面向对象测试负担转移给了集成测试。此外，这种方法还放弃了封装的优点

以类为单元有多种优点（Robert Binder指出，大多数面向对象研究人员和实践者都持这种观点）。在UML语境下，类和描述其行为的“状态图”关联起来。稍后读者就会认识到，这对测试用例标识极为有用。第二种优点是，面向对象集成测试有更清晰的目标，例如，可以检查已经通过单独测试的类的协同操作，这与传统软件测试类似。

16.2 合成与封装的涵义

合成（与分解相反）是面向对象软件开发的核心设计策略。结合重用目标，合成具有非常强烈的单元测试需求，由于单元（类）可以由以前不知道的其他单元合成，因此传统的耦合和聚合概念不再适用。封装有解决这种问题的潜力，但是只有当单元（类）高度内聚，且耦合非常松时，封装才能发挥作用。合成的主要意义是，即使进行了非常好的单元级测试，真正负担还是集中在集成测试层次上。

以下通过例子说明这一点。我们从面向对象角度重新研究土星牌挡风玻璃雨刷系统。很有可能找出三个类：控制杆、雨刷和刻度盘，其行为如图16-1所示。

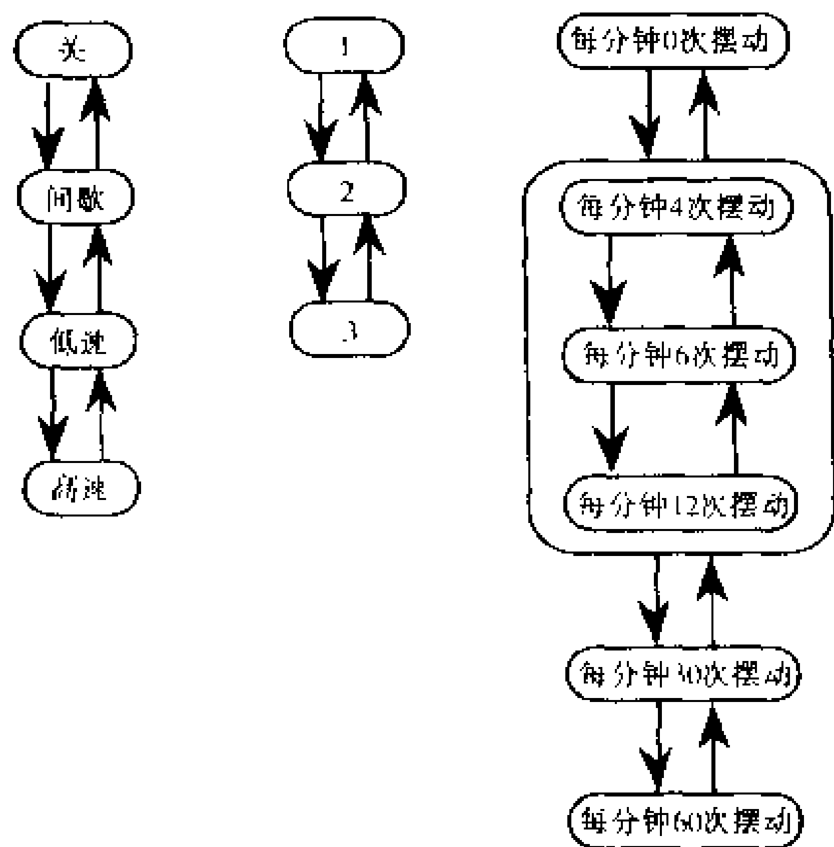


图16-1 挡风玻璃雨刷类的行为

这些类的接口伪代码可以是：

```
Class lever(leverPosition;
        private senseLeverUp(),
        private senseLeverDown() )

Class dial(dialPosition,
        private senseDialUp(),
        private senseDialDown()
```

```
Class wiper(wiperSpeed;
            setWiperSpeed(newSpeed) )
```

控制杆和刻度盘类有感知其相关设备物理事件的操作。当执行这些方法（对应于那些操作）时，会向雨刷类输出相应的设备位置。我们对挡风玻璃雨刷例子感兴趣的部分是，控制杆和刻度盘都是独立设备，并且在控制杆位于“间歇”位置时有交互。封装带来的问题是：这种交互应该在什么地方控制？

封装规则的要求是，类只了解自己的信息，并根据这些信息进行操作。因此，控制杆不知道刻度盘的位置，刻度盘也不知道控制杆的位置。问题是雨刷需要既知道控制杆的位置，也知道刻度盘的位置。正如前面的接口所示，一种可能性是控制杆和刻度盘永远报告各自的位置，由雨刷考虑要做什么。采用这种方法，雨刷类成为“主程序”，并包含整体系统的基本逻辑。

另一种方法是使控制杆类成为“聪明”对象，因为它知道什么时候处于“间歇”位置。采用这种方法，当对控制杆位于“间歇”状态的事件做出响应时，有一个方法获取刻度盘状态（通过getDialPosition（获得刻度盘位置）消息），并直接告诉雨刷以什么速度摆动。通过这种方法，三个类更密切地结合在一起，结果是可重用性较差。这种方法的另一个问题是，如果控制杆处于“间歇”位置，而且以后出现刻度盘事件，这时会发生什么情况呢？控制杆没有理由得到新刻度盘位置信息，也不会向雨刷类发送新消息。

```
Class lever(leverPosition;
            private senseLeverUp(),
            private senseLeverDown() )

Class dial(dialPosition;
            private senseDialUp(),
            private senseDialDown()
            getDialPosition) )

Class wiper(wiperSpeed;
            setWiperSpeed(newSpeed) )
```

第三种方法是建立雨刷主程序（与第一种方法相同），但是使用控制杆和刻度盘类之间的“拥有”关系。采用这种方法，雨刷类使用控制杆和刻度盘类的感知操作检测物理事件。这要求雨刷类要一直以轮询方式保持活动，以便能够观察到控制杆和刻度盘的异步事件。

下面从合成和封装角度讨论这三种方法。第一种方法（叫它第一很贴切，因为这种方法是最好的）在类之间的耦合非常少，可以最大限度地提高重用类的潜力（即以不可预见的方式合成）。例如，更便宜的挡风玻璃雨刷可能会完全忽略刻度盘，而更昂贵的挡风玻璃雨刷可能会用“连续型”刻度盘替代三档刻度盘。对控制杆也可能做类似的改变。在另外两种方法中，类之间的耦合程度较强，会降低合成能力。我们的结论是：很好的封装会得到能够更容易的合成（并因此而重用）和测试的类。

16.3 继承的涵义

虽然将类作为单元看起来很自然，但是继承角色使这种选择变得很复杂。如果给定类继承上层类的属性和/或操作，则单元要满足独立编译指标。Binder提出"扁平类"作为解决这种问题的一种方法（Binder，1996）。扁平类经过扩充以包括全部所继承的属性和操作的原始类。扁平类有些类似结构化分析中被彻底扁平化了的数据流图。（请注意，扁平类由于多重继承而变得复杂，有选择继承和多重有选择继承会使其特别复杂。）扁平类的单元测试可解决继承问题，但是会带来另一个问题。扁平类不是最终系统的一部分，因此有一定的不确定性。此外，扁平类方法对于类的测试是不充分的，下一轮测试要增加特殊用途的测试方法。这样可以为以类为单元的测试提供方便，但是会产生最后一个问题：带有测试方法的类不是（或不应该是）交付系统的一部分。这与在传统软件中，是测试原始代码还是经过处理的代码的问题非常类似。还会引入一些歧义：测试方法也可能是有缺陷的。如果测试方法错误地报告缺陷怎么办？这会产生由方法测试其他方法的没有穷尽的链，非常类似于试图提供形式系统一致性的外部证明。

图16-2显示的是前面提到过的简单自动柜员机（SATM）系统部分的UML继承图，并增加了一些功能，以使这个例子更完善。支票和储蓄账户都有账户编号和余额，并都可以被访问和修改。支票账户有按每张支票收取的费用，必须从支票账户中扣除。储蓄账户有利息，必须定期计算并加入余额。

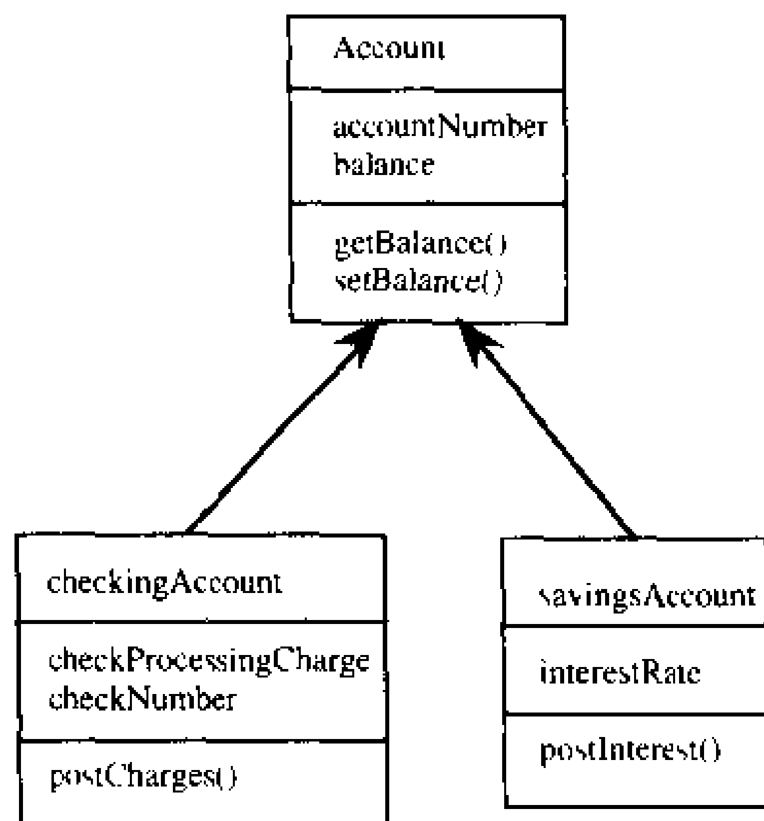


图16-2 类继承

如果不"扁平化"checkingAccount和savingsAccount类就不能访问余额属性，也不能访问或修改余额，对于单元测试来说，这显然是不能接受的。图16-3给出了已经"扁平化"了的checkingAccount和savingsAccount类。这些类显然是适合测试的独立单元，解决了这个问题

又引出另一个问题：通过这种方式会测试getBalance和setBalance两次，因此会在一定程度上损失所期望的面向对象的经济性。

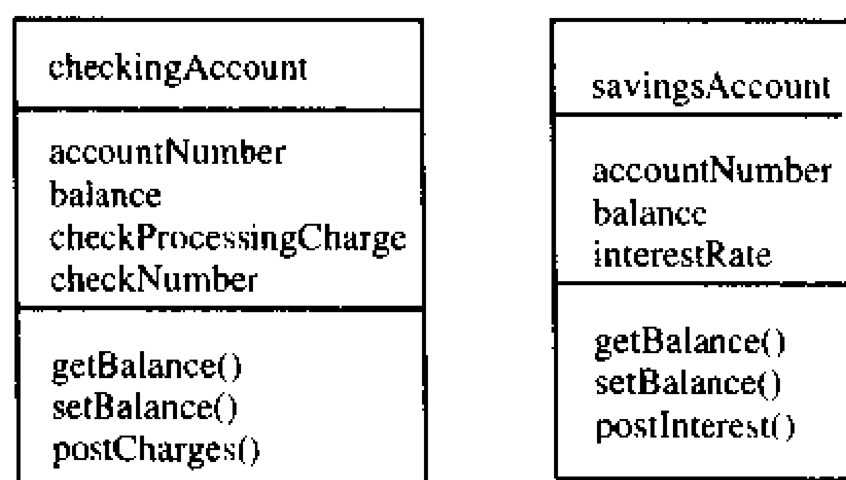


图16-3　被扁平化了的checkingAccount和savingsAccount类

16.4　多态性的涵义

多态性的本质，是同样的方法应用于不同的对象。把类看做单元，意味着多态性的所有问题都要被类/单元测试覆盖。同样，测试多态性操作的冗余性也要考虑所期望的经济性要求

16.5　面向对象测试的层次

取决于单元的构成，面向对象测试采用三层或四层方式。如果把单个操作或方法看做单元，则有四层测试，即操作/方法、类、集成和系统测试。采用这种方法，操作/方法测试与过程性软件的单元测试相同。类和集成测试可以被重命名为类内的测试和类间的测试。第二层包括已经通过测试的操作/方法之间的交互测试。我们认为是面向对象测试主要问题的集成测试，必须考虑已经通过测试的类之间的测试交互。最后，系统测试在端口事件层上进行，并且与（或应该与）传统软件的系统测试相同，惟一不同是系统级测试用例的来源

16.6　GUI测试

由于图形用户界面已经与面向对象软件的关联非常密切，因此我们要用专门一章来讨论这类测试问题。GUI是事件驱动系统的特例，而事件驱动系统的弱点是存在无限事件序列问题。我们将会看到，合理的覆盖概念与表示这些系统的方式密切相关。

16.7 面向对象软件的数据流测试

第10章在讨论数据流测试时，将测试限定在单一单元内。继承和合成问题需要更深入的研究。面向对象测试界目趋一致的观点是，数据流的某种扩展“应该”解决这些特殊需要。第10章已经讨论过，数据流测试的基础是标识单元程序图中的定义和使用节点、然后再考虑各种定义/使用路径。传统软件中的过程调用使这种方法复杂化，常见的做法是将被调用的过程嵌入到要测试的单元中（非常类似被充分扁平化了的类）。第18章将讨论一种事件驱动Petri网的改进版本，能够确切地描述面向对象操作之间的数据流。在这种方法内部，可以描述数据流测试的面向对象扩展。

16.8 第五部分采用的例子

本书在讨论面向对象测试时要使用两个基本例子。第一个是读者已经熟悉了的NextDate问题的面向对象实现；第二个例子是典型的GUI应用程序，即货币转换应用程序。

16.8.1 面向对象的日历

o-oCalendar程序是第2章中NextDate问题例子的一种面向对象实现。图16-4和图16-5给出了o-oCalendar问题的UML类图和类的继承及聚合。伪代码语句行的字母和数字，将在第17章讨论面向对象软件结构测试时使用。

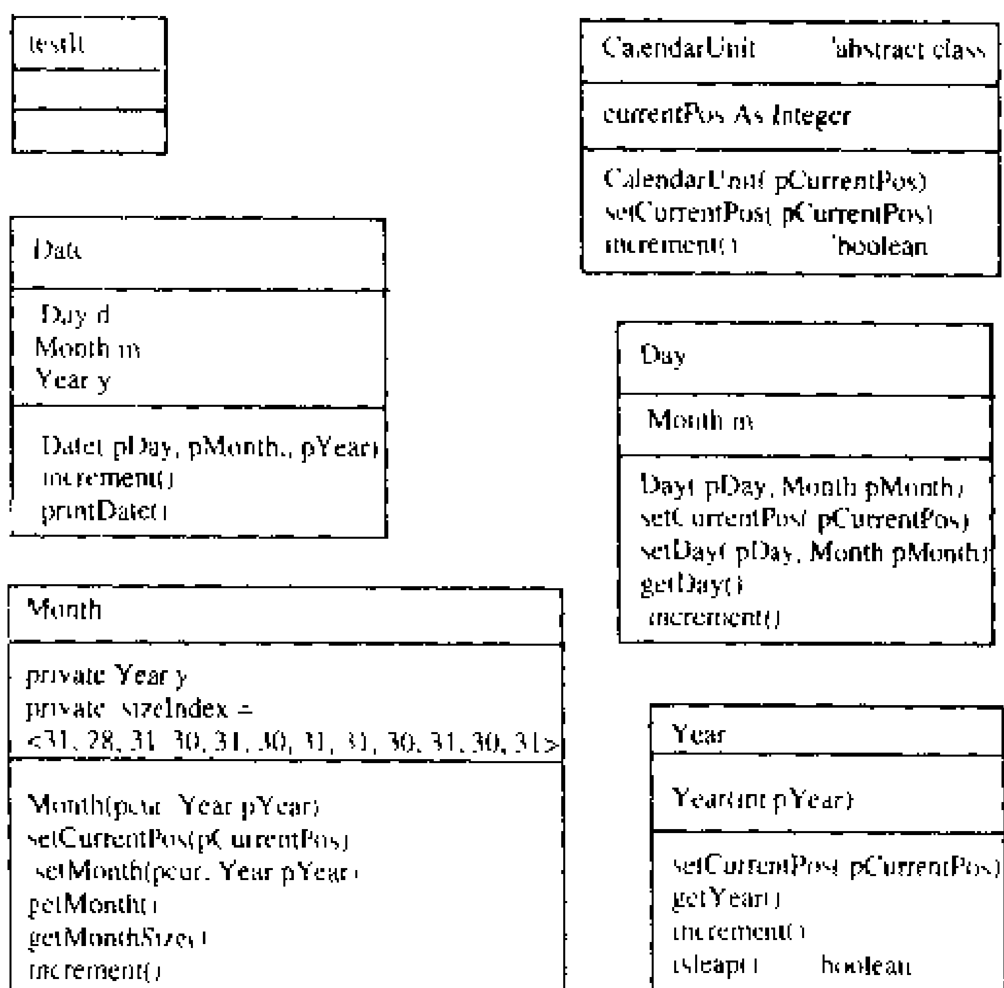


图16-4 o-oCalendar中的类

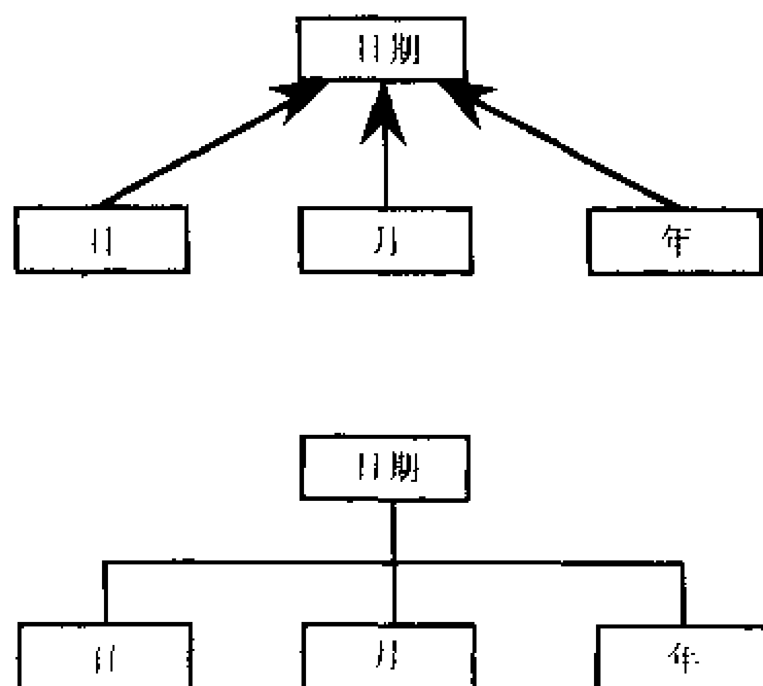


图16-5　类继承与聚合

日期、月份和年类的对象实例，通过抽象类calendarUnit继承集合和增量方法。这些类的每一个都只封装自己所需的信息，其他类的信息都通过消息获得。testIt类实例化一个测试日期，然后对其加1，并打印出结果日期。由于date对象由月份、日期和年实例组成，因此这些对象也被实例化。Date.increment方法向day对象发送消息，查看其是否能够增1（月末问题）。Day.increment方法向month对象发送消息，查看那个月中有几天。然后Date.increment方法向month对象发送消息，查看其是否可以增1（年末问题）。o-oCalendar的伪代码将在第17章中给出。

16.8.2　货币转换应用程序

货币转换程序是一个事件驱动程序，强调与图形用户界面关联的代码。图16-6给出的是采用Visual Basic构建的样本GUI（与第2章中的相同）。

货币转换器

美元金额

等价于……

○ 巴西

○ 加拿大

○ 欧共体

○ 日本

计算

清除

退出

图16-6　货币转换器GUI

这个应用程序将美元转换为以下四种货币中的任意一种：巴西瑞尔、加元、欧元和日元。通过无线电按钮（Visual Basic选项按钮）控制货币选择，这些按钮之间相互排斥。当用户选择了国家之后，系统通过使标签完整做出应答，例如如果选择的是加拿大按钮，则“等于……”就会变成“等于加元”。此外，一面很小的加拿大国旗会出现在等价金额输出位置的旁边。在货币选择之前或之后，用户输入美元金额。两项工作完成之后，用户可按下“计算”按钮、“清除”按钮或“退出”按钮。按下“计算”按钮，可将美元金额转换为所选货币的等价金额。按下“清除”按钮，可重新设置货币选择、美元金额、等价货币金额和相关标签。按下“退出”按钮，结束该应用程序。

为了测试GUI应用程序，首先要标识所有用户输入事件和所有系统输出事件（必须是外部可视、可观察的）。货币转换程序的这些事件如表16-1所示。

表16-1 货币转换程序的输入和输出事件

	输入事件		输出事件
ip1	输入美元金额	op1	显示美元金额
ip2	按下国家按钮	op2	显示货币名称
ip2.1	按下巴西	op2.1	显示巴西瑞尔
ip2.2	按下加拿大	op2.2	显示加元
ip2.3	按下欧共体	op2.3	显示欧元
ip2.4	按下日本	op2.4	显示日元
ip3	按下“计算”按钮	op2.5	显示省略号
ip4	按下“清除”按钮	op3	指示所选国家
ip5	按下“退出”按钮	op3.1	指示巴西
ip6	在错误消息中按下OK	op3.2	指示加拿大
		op3.3	指示欧共体
		op3.4	指示日本
		op4	重新设置所选国家
		op4.1	重新设置巴西
		op4.2	重新设置加拿大
		op4.3	重新设置欧共体
		op4.4	重新设置日本
		op5	显示外币值
		op6	错误消息：必须选择国家
		op7	错误消息：必须输入美元金额
		op8	错误消息：必须选择国家并输入美元金额
		op9	重新设置美元金额
		op10	重新设置等价货币金额

图16-7给出了该应用程序高层、几乎完整的视图。状态是外部可视的GUI外观。例如，当应用程序启动后，会出现空闲状态，没有用户输入（如图16-6所示）。这种情况还会出现在按下“清除”按钮事件之后。当用户输入完美元金额并且没有做其他事时，出现美元金

额输入状态。其他状态也有类似名称。请注意，选择国家状态实际上是宏状态，只选择了四个国家中的一个。这个状态在图16-8中显示得更详细。转移上的标注是表16-1中输入和输出事件的缩写。令人吃惊地是，即使这样一个小小的GUI也那么复杂。通过使用人工输入事件，例如ip2：按钮国家按钮，可以使复杂性得到一定缓解。请注意，假想输入事件ip2和假想输出事件op2、op3和op4，大大简化了图16-7所示的高层有限状态机（FSM）。这样做省略了一些事件，尤其是按下“清除”和“退出”按钮事件。这些事件可以在任何状态中发生，显示这些事件会使框图很散乱。

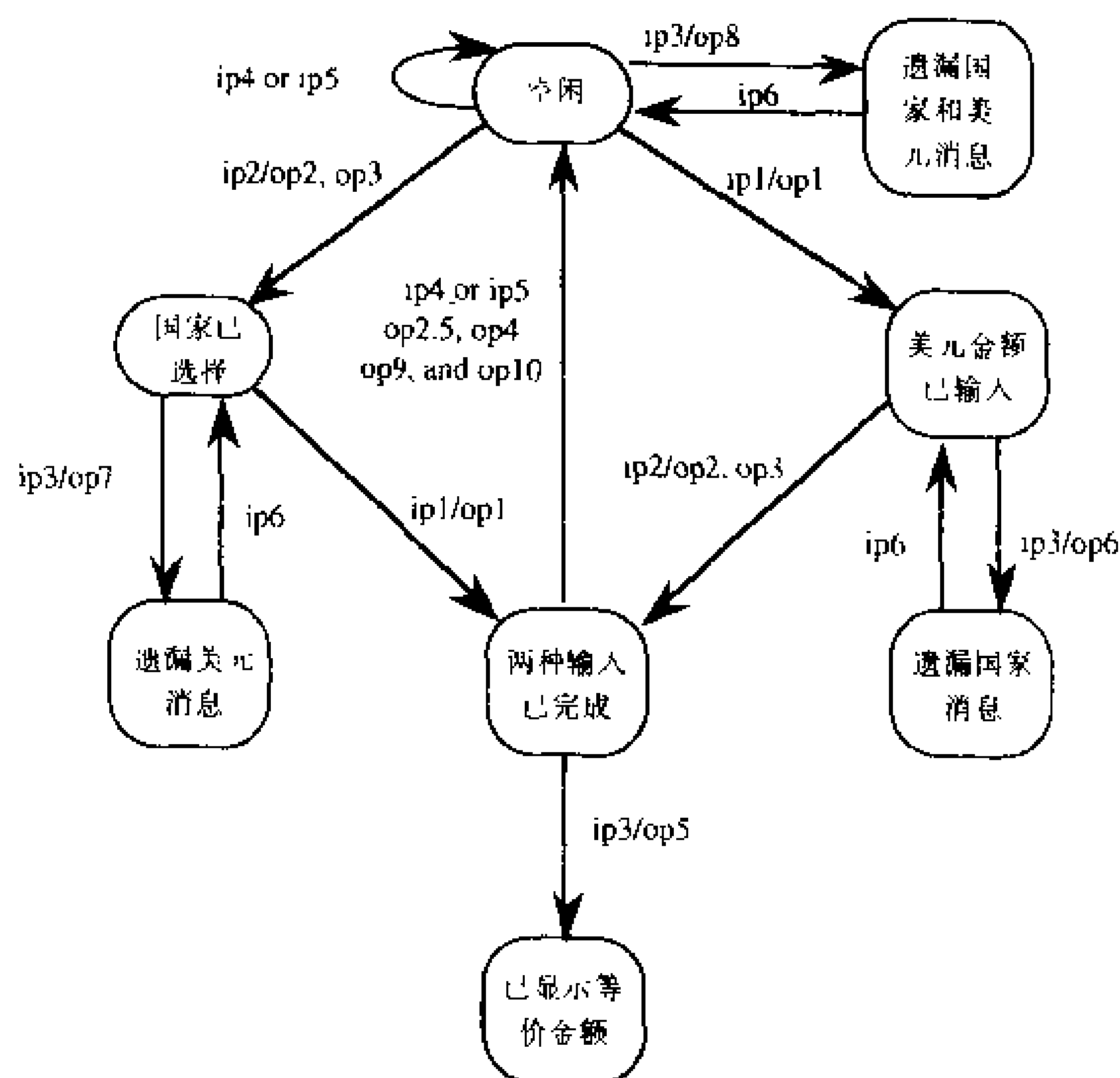


图16-7 高层有限状态机

图16-8给出了选择国家状态内部行为的详细视图。图中维持国家按钮之间的异或关系。图16-8中更详细的视图用相应的多个实际事件替代了人工输入和输出事件。但是图16-8是不完整的。巴西和日本之间、加拿大和欧共体之间的转移标注没有给出。此外，图16-7和图16-8都没有显示在每个状态会发生的所有事件，尤其是按下“清除”或“退出”按钮事件。

熟悉“状态图”表达法的读者会熟悉将这种重复复杂性由表示法表示的很好方法。图16-9更完整地表达了图16-7的高层视图。请注意，从“执行”状态到“存储”状态的转移，显示的是用户输入ip6以退出应用程序，操作系统“结束应用程序”事件可以在执行状态内部的任何状态中出现。类似地，“清除”输入事件（ip4）可以出现在未命名状态内部的任何状态中。

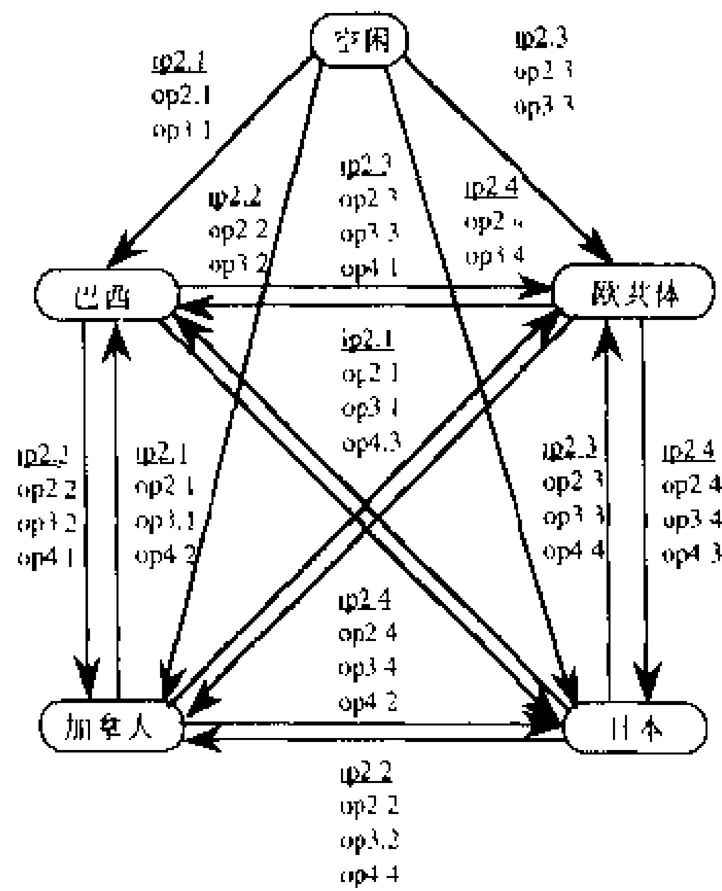


图16-8 选择国家状态详细视图

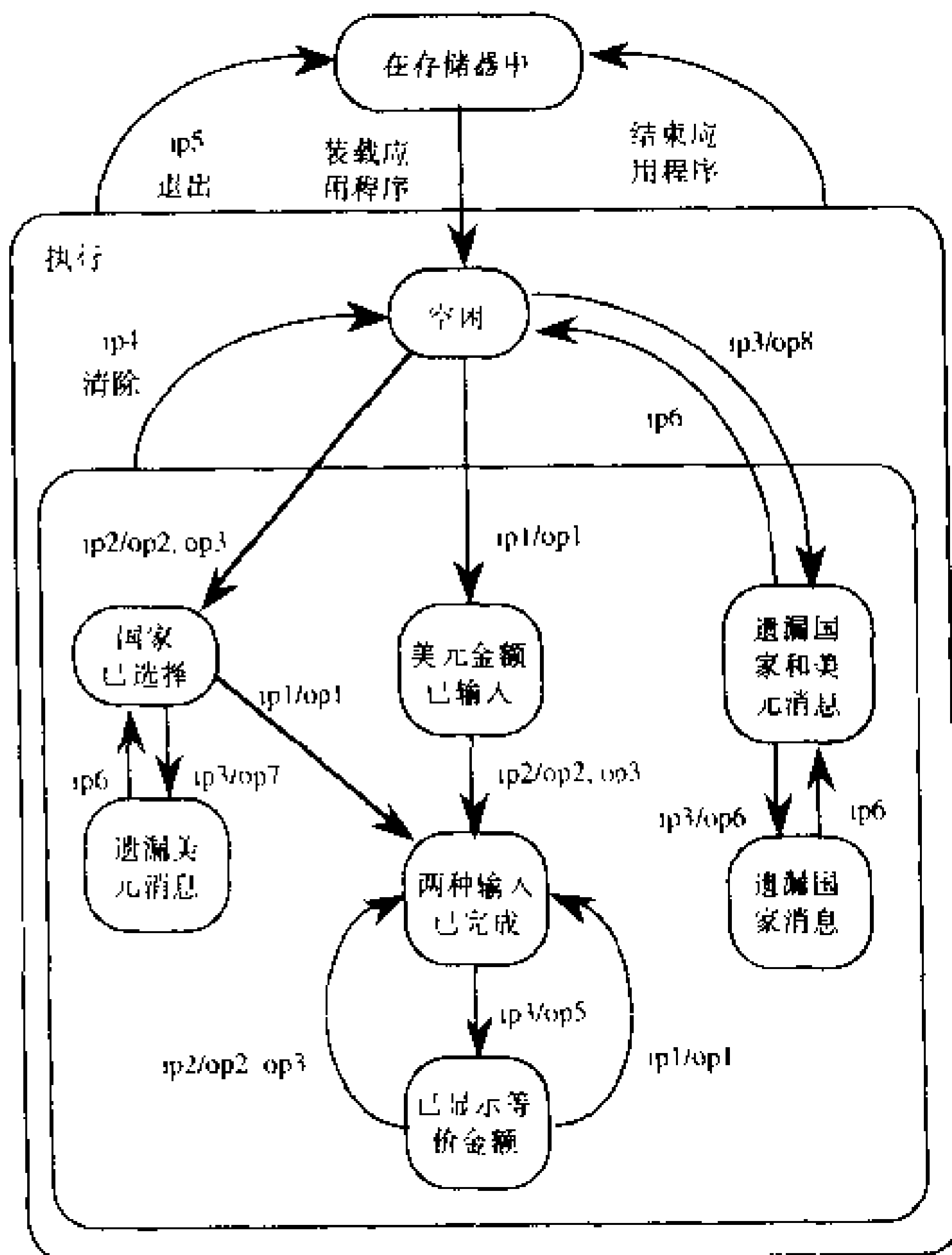


图16-9 货币转换器应用程序的状态图

处理被精简事件的一种方法是采用单个状态层框图显示对每种可能的输入事件的应答。采用事件表也可以很好地做到这一点。表16-2给出的是通过这种方法对“完成两种输入”状态的处理。

表16-2　“完成两种输入”状态的事件表

	输入事件		输出事件
ip1	输入美元金额	op1	显示美元金额
ip2.1	按下巴西	op2.1	显示巴西瑞尔
		op3.1	指示巴西
ip2.2	按下加拿大	op2.2	显示加元
		op3.2	指示加拿大
ip2.3	按下欧共体	op2.3	显示欧元
		op3.3	指示欧共体
ip2.4	按下日本	op2.4	显示日元
		op3.4	指示日本
ip3	按下“计算”按钮	op5	显示外币值
ip4	按下“清除”按钮	op2.5	显示省略号
		op4	重新设置所选国家
		op9	重新设置美元金额
		op10	重新设置等价货币金额
ip5	按下“退出”按钮		应用程序结束忽略（这个
ip6	在错误消息中按下OK		状态中的错误消息不可视）

16.9 参考文献

Binder, Robert V., The free approach for testing use cases, threads, and relations, *Object*, Vol. 6, No. 2, February 1996.

Kiczales, Gregor et al., Aspect-oriented programming. Proceedings of the European Conference on Object-Oriented programming (ECOOP), Finland, Springer-Verlag, LNCS1241, June 1997.

16.10 练习

可以通过多种方式扩展o-oCalendar问题。一种扩展是添加占星术内容：黄道十二宫中的每一个都有名称和开始日期（每月的21日）。请在月份类中添加属性和方法，使得testIt可以找出给定日期的黄道十二宫。

第17章 类测试

类测试的主要问题是，类和方法哪一个是单元。大多数面向对象文献都倾向于认为类是单元，但是这种定义存在问题。在传统软件（也很难找到单元的定义）中，单元的常见指导方针是：

能够自身编译的最小程序块。
单一过程/函数（独立）。
由一个人完成的小规模工作。

这些指导方针对于面向对象软件同样有意义，但是都没有回答究竟是把类还是方法作为单元。一个方法实现一个功能，不会指派给多个人开发，因此可以合法地将方法看做单元最小编译要求存在问题。从技术上看，我们可以忽略类中的其他方法（可以将这些方法注释掉），但是这会带来组织上的混乱。本书将给出两种面向对象单元测试观点，读者可以根据具体环境确定最合适的方法。

17.1 以方法为单元

简单地说，这种方法可以将面向对象单元测试归结为传统的（过程的）单元测试，方法几乎等价于过程，所以可以使用所有传统功能性测试和结构性测试技术。过程代码的单元测试需要桩和驱动器测试程序，以提供测试用例并记录测试结果。类似地，如果把方法看做是面向对象单元，也必须提供能够实例化的桩类，以及起驱动器作用的“主程序”类，以提供和分析测试用例。

如果更仔细地研究单个方法，就会发现令人高兴的封装结果：方法一般很简单。在后面的讨论中，请找出构成o-oCalendar应用程序类的伪代码和对应的程序图。请注意，圈复杂度总是很低，Date类的增量方法圈复杂度最高，也只有V（G）=3。公平地说，这个实现特意设计得很简单，不需要对有效输入进行检查。如果要检验有效输入，则圈复杂度就会提高。

即使圈复杂度很低，但是接口复杂度仍然很高，再看一下Date.increment，请注意高密度的消息传送：消息要发送给Day类的两个方法、Year类的一个消息和Month类的两个方法中，这意味着创建合适桩的工作量，差不多与标识测试用例的工作量相同。另一个更重要的结果是，大部分负担被转移到集成测试中。事实上，我们可以标识两级集成测试，即类内集成测试和类间集成测试。

17.1.1　o-oCalendar的伪代码

UML在单元/类级需要很少文档。以下我们为每个类增加类责任协同（CRC）卡，后面是类的伪代码，然后是类操作的程序图（如图17-1、17-2、17-3和17-4所示）

17.1.1.1　类：CalendarUnit

责任：提供一个操作在所继承的类中设置取值，提供一个布尔操作说明所继承类中的属性是否可以增1。

Collaborates with: Day, Month, and Year

```
    class CalendarUnit 'abstract class
    currentPos As Integer

    CalendarUnit (pCurrentPos)
        currentPos = pCurrentPos
    End     'CalendarUnit

a.  setCurrentPos(pCurrentPos)
b.      currentPos = pCurrentPos
    End     'setCurrentPos

    abstract protected boolean increment()
```

17.1.1.2　类：testIt

责任：用做测试驱动器，即创建一个测试日期对象，然后请求该对象对其本身增1，最后打印新值。

Collaborates with: Date

```
    class testIt
    main()
1.      testdate = instantiate Date(testMonth, testDay, testYear)
2.      testdate.increment()
3.      testdate.printDate()
    End     'testIt
```

17.1.1.3　类：Date

责任：Date对象由日期、月份和年对象组成，Date对象使用所继承的Day和Month对象中的布尔增量方法对其本身增1。如果日期和月份对象本身不能增1（例如月份或年的最后一天），则Date的增量方法会根据需要重新设置日期和月份。如果是12月31日，则年也要增1。printDate操作使用Day、Month和Year对象中的get()方法，并以mm/dd/yyyy格式打印出日期。

Collaborates with: testIt, Day, Month, and Year

```
    class Date
    private Day d
    private Month m
    private Year y
```

```
4.     Date(pMonth, pDay, pYear)
5.         y = instantiate Year(pYear)
6.         m = instantiate Month(pMonth, y)
7.         d = instantiate Day(pDay, m)
       End     'Date constructor

8.     increment ()
9.         if (NOT(d.increment()))
10.    Then
11.        if (NOT(m.increment()))
12         Then
13.            y.increment()
14.            m.setMonth(1,y)0
15.        Else
16.            d.setDay(1, m)
17.        EndIf
18.    EndIf
       End     'increment

19  printDate ()
20.     Output (m.getMonth() + "/" + d.getDay() + "/" + y getYear())
    End     'printDate
```

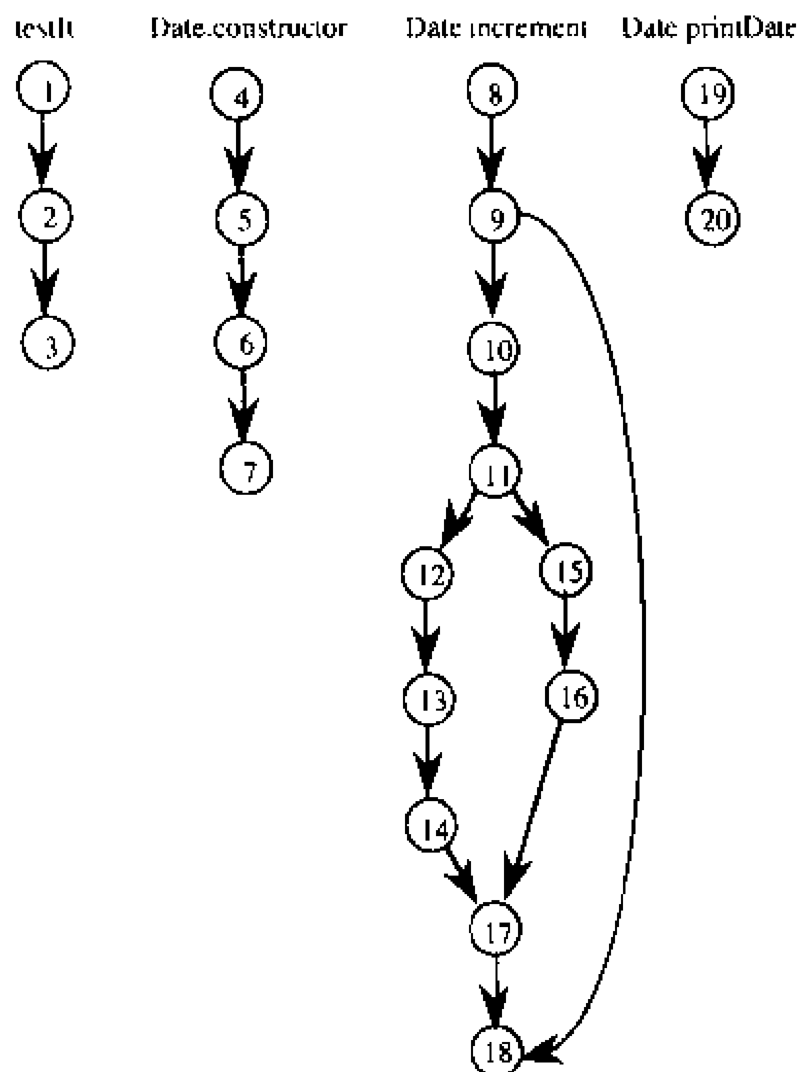


图17-1 testIt和Date类的程序图

17.1.1.4 类：Day

责任：Day对象有一个私有月份属性，增量方法用来查看日期取值是要增1还是复位为1。

Day对象也提供get()和set()方法。

Collaborates with: Month

```
class Day isA CalendarUnit
private Month m

21. Day( pDay, Month pMonth)
22.     setDay( pDay, pMonth)
        End    'Day costructor

23. setDay( pDay, Month pMonth)
24.     setCurrentPos(pDay)
25.     m = pMonth
        End    'setDay

26. getDay()
27.     return currentPos
        End    'getDay

28. boolean increment()
29.     currentPos = currentPos + 1
30.     if (currentPos <= m.getMonthSize())
31.         Then    return true
32.         Else    return false
33.     EndIf
        End    'increment
```

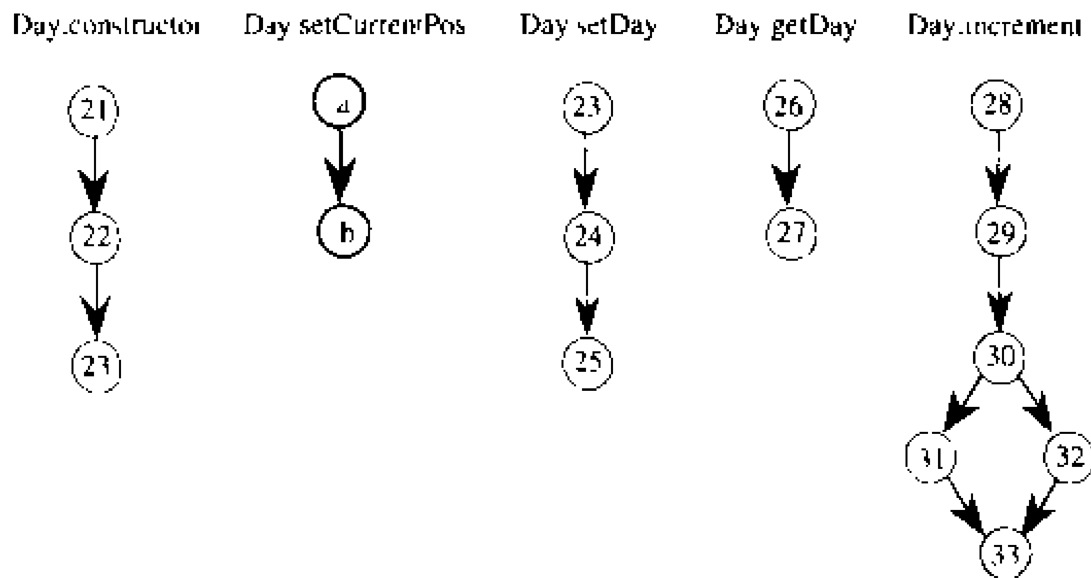


图17-2 Day类的程序图

17.1.1.5 类：Month

责任：Month对象有一个值属性，用做月份最后一日值的数组下标（例如1月的最后一日是31，2月的最后一日是28，等等）。Month对象提供get()和set()服务，以及所继承的布尔增量方法。2月29日的判决，由给Year对象的isLeap消息做出。

Collaborates with: Year

```
class Month isA CalendarUnit
    private Year y
```

```
        private sizeIndex = <31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31>

34. Month( pcur, Year pYear)
35.     setMonth( pCurrentPos, Year pyear)
        End     'Month constructor

36  setMonth( pcur, Year pYear)
37.     setCurrentPos(pcur)
38.     y = pYear
        End     'setMonth

39. getMonth()
40.     return currentPos
        End     'getMonth

41. getMonthSize()
42.     if (y.isleap())
43.         Then    sizeIndex[1] = 29
44.         Else        sizeIndex[1] = 28
45.     EndIf
46.     return sizeIndex[ currentPos -1]
        End     'getMonthSize

47. boolean increment()
48.     currentPos = currentPos + 1
49.     if (currentPos > 12)
50.         Then    return false
51.         Else    return true
52.     EndIf
        End     'increment
```

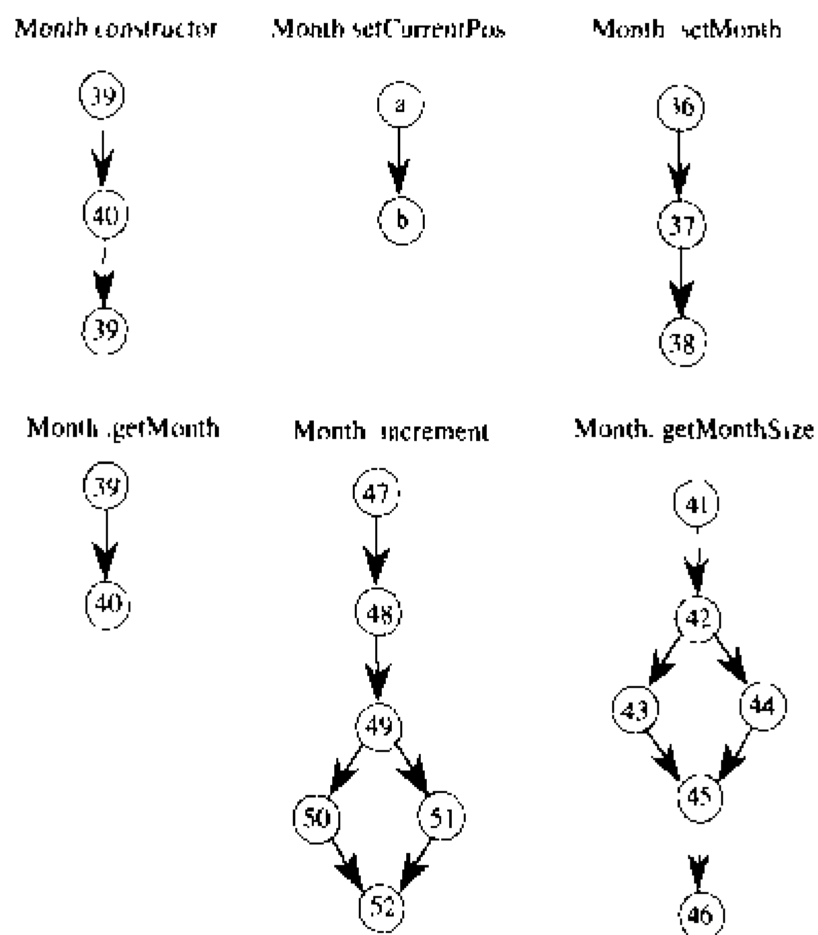


图17-3 Month类的程序图

17.1.1.6 类：Year

责任：除了一般的get()和set()方法之外，如果测试日期是任意年的12月31日，Year对象自己也要增1。Year对象提供一个布尔服务，说明当前值是否对应闰年。

```
Collaborates with: (no external messages sent)

    class Year isA CalendarUnit
53. Year( pYear)
54.       setCurrentPos(pYear)
          End      'Year constructor

55. getYear()
56        return currentPos
          End      'getYear

57. boolean increment()
58.       currentPos = currentPos + 1
59.       return true
          End      'increment

60. boolean isleap()
61.       if (((currentPos MOD 4 = 0) AND NOT(currentPos MOD 400 = 0)) OR
             (currentPos MOD 400 = 0))
62.          Then    return true
63.          Else    return false
64.          EndIf
          End      'isleap
```

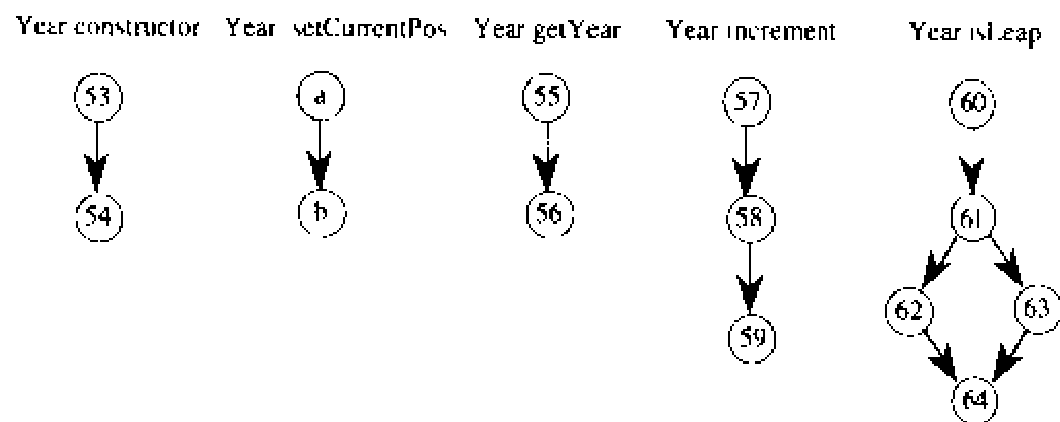


图17-4 Year类的程序图

17.1.2 Date.increment的单元测试

正如第6章所介绍的，等价类测试是逻辑密集单元的好选择。Date.increment操作处理日期的三个等价类：

D1 = {日期：1 <= 日期 < 月的最后日期}

D2 = {日期：日期是非12月的最后日期}

D3 = {日期：日期是12月31日期}

首先，这些等价类看起来是松散定义的，尤其是D1，引用了没有月份的说明的最后日期，没有引用哪个月份。幸亏有封装，我们才可以忽略这些问题。（实际上问题被转化为Month.increment操作的测试。）

17.2 以类为单元

把类作为单元可以解决类内集成问题，但是会产生其他问题。一个问题与类的各种视图有关。在静态视图中，类作为源代码存在。如果我们要做的只是代码读出技术，则这没有什么问题。静态视图的问题是继承被忽略，但是通过被充分扁平化了的类可以解决这个问题。可以把第二种视图称做编译时间视图，因为继承实际"发生"在编译时。第三种视图是执行时间视图，但是仍然有一些问题，例如不能测试抽象类，因为不能被实例化。此外，如果使用充分扁平化的类，则还要在单元测试结束后，将其"恢复"为原来的形式。如果不使用充分扁平化的类，则为了编译类，需要在继承树中高于它的所有其他类。可以想像，软件配置管理也有这种需求。

把类作为单元，在没有什么继承，并且类具有我们称之为内部控制复杂性时最有意义。类本身应该拥有一种"有意思"（与简单或枯燥相反）的状态图，并且应该有相当多的内部消息传递。为了讨论以类为单元的测试，以下重新研究挡风玻璃雨刷例子的更复杂的版本。

17.2.1 windshieldWiper类的伪代码

第16章讨论过的三个类，在这里被合并为一个。通过这种方式，方法感知控制杆和刻度盘，并在leverPosition和dialPosition状态变量中维护控制杆和刻度盘的状态。当出现控制杆或刻度盘事件时，相应的感知方法会向setWiperSpeed方法发送一个（内部）消息，由setWiperSpeed方法设置相应地设置状态变量wiperSpeed。经过修改的windshieldWiper类有三个属性，每个变量的获取和设置操作，以及感知控制杆和刻度盘设备四种物理事件的方法。

```
class windshieldWiper

    private wiperSpeed
    private leverPosition
    private dialPosition

    windshieldWiper(wiperSpeed, leverPosition, dialPosition)

    getWiperSpeed()
    setWiperSpeed()

    getLeverPosition()
    setLeverPosition()

    getDialPosition()
    setDialPosition()

    senseLeverUp()
```

```
senseLeverDown()

senseDialUp()
senseDialDown()

End class windshieldWiper
```

类行为如图17-5中的“状态图”所示，其中三个设备出现在正交组件中，在Dial和Lever组件中，转移由事件引起，因此雨刷组件中的转移，都是由与Dial和Lever正交组件中“活动”状态有关的命题引起的。（这类命题是StateMate产品使用的转移表示方法丰富句法的一部分。）

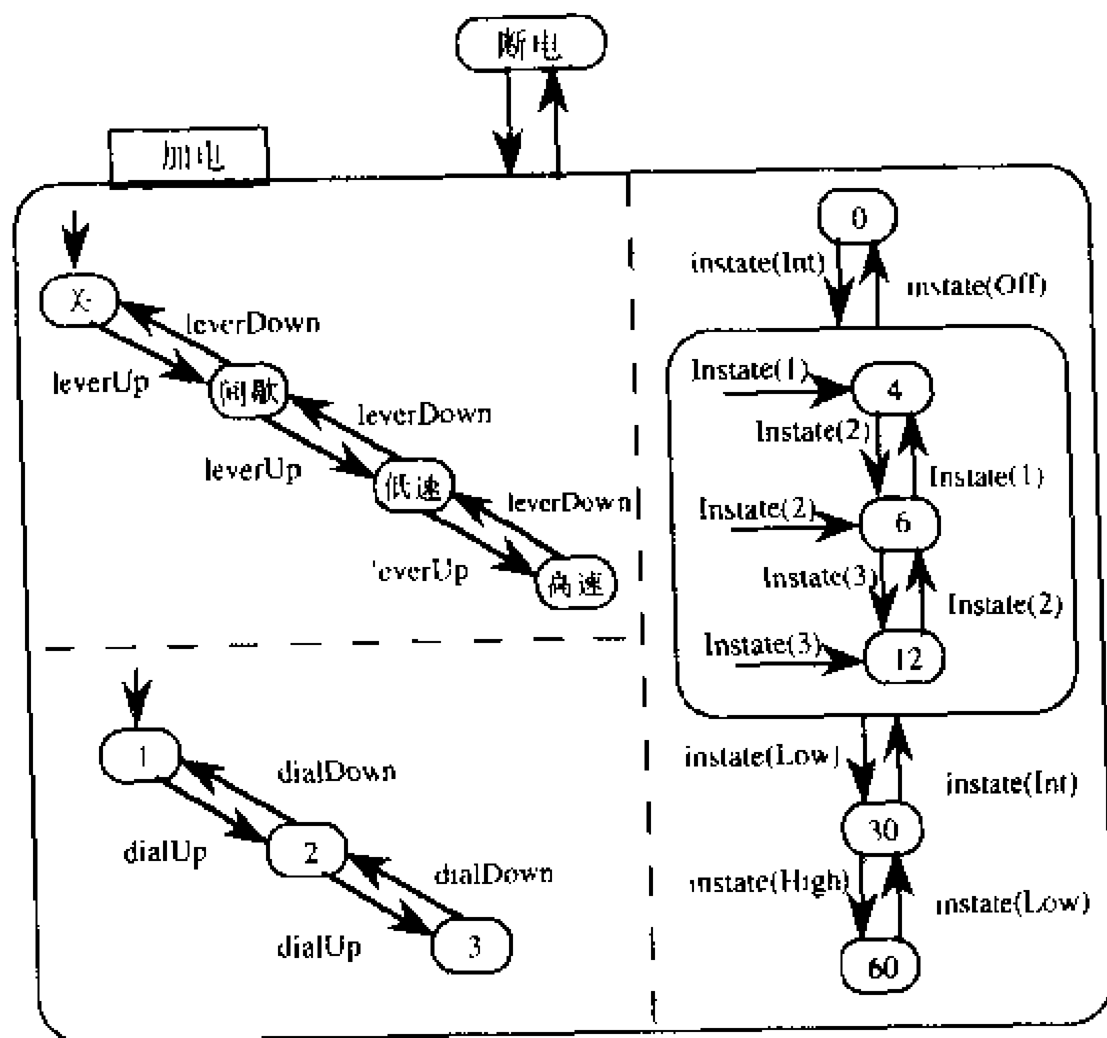


图17-5 windshieldWiper类的“状态图”

17.2.2 windshieldWiper类的单元测试

以类为单元选择的困难部分，是因为有单元测试的层次。在我们的例子中，自底向上从状态变量的获取/设置方法开始的测试是有意义的（只有当其他类需要这些方法时才提出来）。刻度盘和控制杆感知方法都非常相似，以下给出senseLeverUp方法的伪代码。

```
senseLeverUp()
    Case leverPosition Of
        Case 1: Off
            leverPosition = Int
            Case dialPosition Of
                Case 1: 1
```

```
                        wiperSpeed = 4
                Case 2: 2
                        wiperSpeed = 6
                Case 3: 3
                        wiperSpeed = 12
            EndCase    'dialPosition
        Case 2: Int
            leverPosition = Low
            wiperSpeed = 30
        Case 3: Low
            leverPosition = High
            wiperSpeed = 60

    Case 4: High
        (impossible; error condition)
EndCase    'leverPosition
```

测试senseLeverUp方法需要Case和嵌套Case语句中的不同分支。"外层"Case语句的测试覆盖状态图中对应的leverUp转移。类似地，我们还必须测试leverDown、dialUp和dialDown方法，一旦我们知道刻度盘和控制杆组件是正确的，就可以测试雨刷组件了

测试驱动器类的伪代码与以下给出的代码类似：

```
class testSenseLeverUp
    wiperSpeed
    leverPos
    dialPos
    testResult    'boolean
main()
    testCase = instantiate windshieldWiper(0, Off, 1)
    windshieldWiper.senseLeverUp()
    leverPos = windshieldWiper.getLeverPosition()
    If leverPos = Int
        Then testResult = Pass
        Else testResult = Fail
    EndIf
End    'main
```

此外还有两个测试用例，用来测试从"间歇"到"低速"和从"低速"到"高速"的转移。下面，采用以下伪代码测试windshieldWiper类的其他部分

```
class testWindshieldWiper
    wiperSpeed
    leverPos
    dialPos
    testResult    'boolean
main()
    testCase = instantiate windshieldWiper(0, Off, 1)
    windshieldWiper.senseLeverUp()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 4
        Then testResult = Pass
        Else testResult = Fail
    EndIf
End    'main
```

这里有两个微妙问题。简单的问题是实例化windshieldWiper语句要建立测试用例的前提。伪代码中的测试用例正好对应图17-5"状态图"的"刻度盘"和"控制杆"组件的默认输入状态。第二个微妙问题比较难理解。状态图的雨刷组件具有我们所谓的类的测试人员（或外部）视图。在这种视图中，雨刷默认输入和转移都是各种inState命题引起的，但是实现通过设置方法改变状态变量的值引起转移。

有了类行为的"状态图"定义之后，可以用来以与有限状态机标识系统级测试用例非常相似的方法来定义测试用例。基于状态图的类测试支持合理的测试覆盖指标。一些明显的指标包括：

每个事件。
组件中的每个状态。
组件中的每个转移。
所有（不同组件中的）交互状态对偶。
对应于客户定义用例的场景。

表17-1给出的是"控制杆"组件"组件中每个状态"覆盖层次的实例化语句（用于建立前提）和预期输出的测试用例。

表17-1　"控制杆"组件的测试用例

测试用例	前提（实例化语句）	windshieldWiper事件（方法）	预期leverPos的输出值
1	WindshieldWiper（0，Off，1）	senseLeverUp（）	"间歇"
2	WindshieldWiper（0，Int，1）	senseLeverUp（）	"低速"
3	WindshieldWiper（0，Low，1）	senseLeverUp（）	"高速"
4	WindshieldWiper（0，High，1）	senseLeverDown（）	"低速"
5	WindshieldWiper（0，Low，1）	senseLeverDown（）	"间歇"
6	WindshieldWiper（0，Int，1）	senseLeverDown（）	"关"

请注意，较高层次的覆盖实际上适用于方法的类内集成，这看起来与以类为单元的思想矛盾。场景覆盖准则与系统级测试几乎相同。以下是一个测试用例以及在一个测试类中所需的相应消息序列。

UC1	正常用法
描述	挡风玻璃雨刷在"关"位置，"刻度盘"在位置1；用户将控制杆推到"间歇"，然后将刻度盘从位置2转到位置3；然后将控制杆推到"低速"；用户将控制杆推到"间歇"，然后推到"关"
前提	挡风玻璃雨刷在"关"位置，"刻度盘"在位置1，雨刷速度为0

事件序列	用户行动	系统应答
1	将控制杆推到"间歇"	雨刷速度为4
2	将刻度盘转到2	雨刷速度为6
3	将刻度盘转到3	雨刷速度为12
4	将控制杆推到"低速"	雨刷速度为20
5	将控制杆推到"间歇"	雨刷速度为12
6	将控制杆推到"关"	雨刷速度为0

```
class testScenario
    wiperSpeed
    leverPos
    dialPos
    step1OK    'boolean
    step2OK    'boolean
    step3OK    'boolean
    step4OK    'boolean
    step5OK    'boolean
    step6OK    'boolean

main()
    testCase = instantiate windshieldWiper(0, Off, 1)
    windshieldWiper.senseLeverUp()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 4
        Then step1OK = Pass
        Else step1OK = Fail
    EndIf

    windshieldWiper.senseDialUp()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 6
        Then step2OK = Pass
        Else step2OK = Fail
    EndIf

    windshieldWiper.senseDialUp()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 12
        Then step3OK = Pass
        Else step3OK = Fail
    EndIf

    windshieldWiper.senseLeverUp()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 20
        Then step4OK = Pass
        Else step4OK = Fail
    EndIf

    windshieldWiper.senseLeverDown()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 12
        Then step5OK = Pass
        Else step5OK = Fail
    EndIf

    windshieldWiper.senseLeverDown()
    wiperSpeed = windshieldWiper.getWiperSpeed()
    If wiperSpeed = 0
        Then step6OK = Pass
        Else step6OK = Fail
    EndIf
End    'main
```

第18章 面向对象的集成测试

在软件测试的三个主要层次中，集成测试是大家理解得最不透彻的，不论是传统软件还是面向对象软件都是这样。与传统过程软件一样，面向对象集成测试也假设已完成单元级测试。对于集成测试，两种单元选择方法都需要进行集成测试 如果采用操作/方法作为单元，则需要进行两级集成：一级是将操作集成到完整类中，另一级是将类与其他类集成。这是不能不做的，将操作作为单元的全部理由就是类太大，并涉及多个设计人员。

下面讨论更常见的以类为单元的方法，一旦完成单元测试，必须执行两个步骤：(1) 如果使用了被扁平化了的类，则必须恢复最初的类层次结构；(2) 如果增加了测试方法，则必须删除。

一旦建立了“集成测试床”，就需要标识应该测试的内容。与传统软件集成一样，可以选择静态和动态测试。可以通过纯静态方式处理由多态性引入的复杂性：测试与每个动态语境有关的消息。面向对象集成测试的动态视图更加重要。

18.1 集成测试的UML支持

本章和下一章将使用统一建模语言（UML）描述例子。有关的UML大量信息可以从www.rational.com网站得到。在采用UML定义的面向对象软件中，协同图和序列图是集成测试的基础。在系统层，UML描述由各种层次的“用例”、“用例”图、类定义和类图组成（如图16-4和16-5所示）。一旦定义了这一层，就增加了集成层细节。协同图显示类之间的（部分）信息传输。图18-1是o-oCalendar应用程序的协同图。协同图非常类似在第13章中曾经使用过的单元调用图（图13-2给出了一个例子）。因此，协同图既支持成对方法，也支持相邻集成测试。

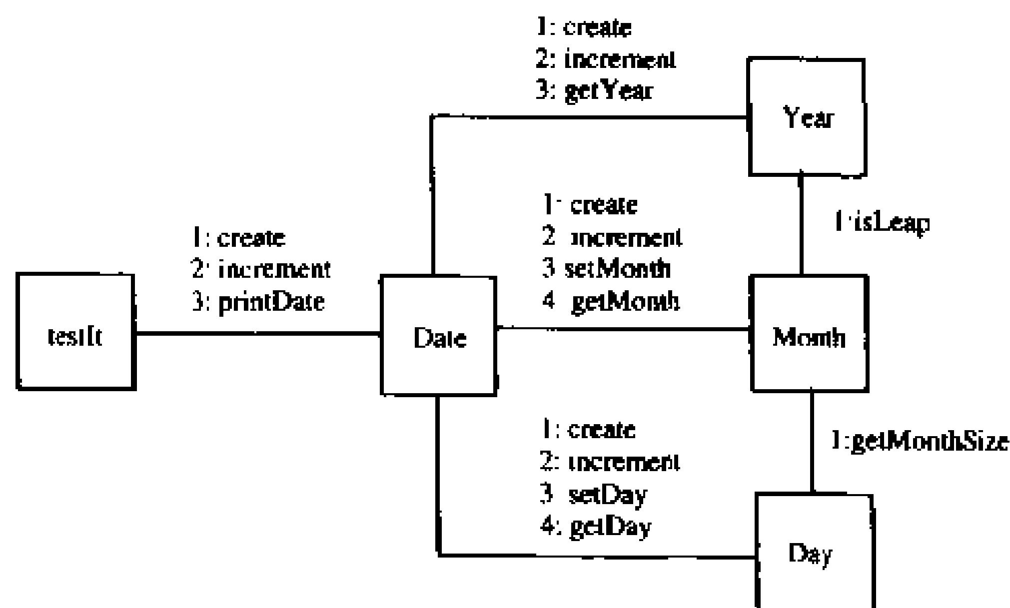


图18-1 o-oCalendar的协同图

对于成对集成，通过向被集成的类发送消息或接收消息的独立“相邻”类进行单元（类）测试，就类向其他类发送/接收消息而言，其他类必须表示为桩。所有这类额外工作都使成对的类集成难以接受，这与过程单元的成对集成一样。根据图18-1所示的协同图，可以得到以下要集成的类对：

testIt和Date，为Year、Month和Day建立桩。
Date和Year，为testIt、Month和Day建立桩。
Date和Month，为testIt、Year和Day建立桩。
Date和Day，为testIt、Month和Year建立桩。
Year和Month，为Date和Day建立桩。
Month和Day，为Date和Year建立桩。

以协同图为基础进行面向对象集成测试的缺点是，在类级，UML所选的行为模型是“状态图”。对于类级行为，“状态图”是测试用例的很好的基础，特别适合以类为单元的测试。不过问题是，一般来说很难合并“状态图”，以便在较高层次上观察行为（Regmi，1999）。

相邻集成会带来图论很有意思的问题。使用图18-1中的（无向）图，Date的邻居是整个图，而testIt的邻居只是Date。结果，数学家标识了各种线性图“中心”。例如，其中的一种是超中心，这种中心最小化到图中其他节点的最大距离。在集成顺序上，可以想像在平静的水面上投入石子所产生的环状涟漪。我们首先从超中心和一条边之外的节点邻居开始，然后再增加两条边之外的节点，等等。类的邻居集成肯定会降低桩工作量，但是要付出降低诊断精度的代价。如果测试用例失败，则必须在更多的类中寻找缺陷。

序列图跟踪通过协同图的执行时间路径。（在UML中，序列图有两级，即系统/用例级和类交互级）。粗竖线表示类或类的实例，箭头标号表示类（的实例）按时间顺序发送的消息。图18-2给出的是打印新日期的部分o-oCalendar应用程序的序列图。

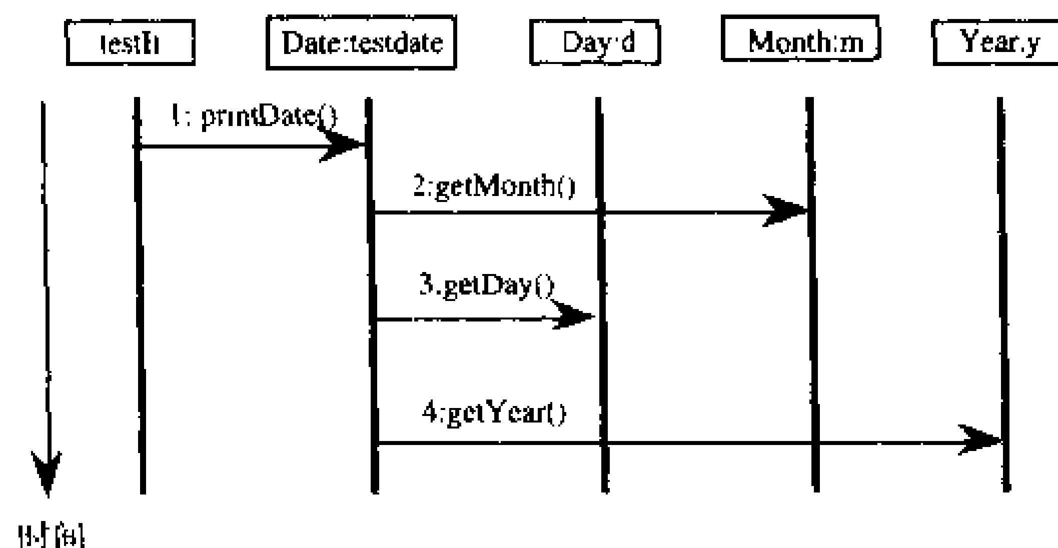


图18-2 printDate的序列图

就我们所创建的序列图来说，是面向对象集成测试的很好基础。这些序列图几乎等价于面向对象版的MM-路径（将在下一节定义）。这种序列图的实际测试应该采用与以下类似的

伪代码：

```
1. testDriver
2.   m.setMonth(1)
3.   d.setDay(15)
4.   y.setYear(2002)
5.   Output ("expected value is 1/15/2002")
6.   Output ("actual output is ")
7.   testIt.printDate()
8. End testDriver
```

语句2、3和4使用已通过单元测试的方法设置要向其发送消息的类的预期输出。从以上程序可以看出，这种测试驱动器要依靠人员根据打印输出做出通过或失败判断。我们可以把比较逻辑放到testDriver类中，以做内部比较。这可能存在问题，因为如果我们在被测代码中会犯错误，那么在比较逻辑中同样也会犯错误。

18.2　面向对象软件的MM-路径

在提到传统软件的MM-路径时，我们用"消息"表示个体单元（模块）之间的调用，采用模块执行路径（模块级线索）取代完整的模块。这里我们使用同样的缩写表示由消息分开的各种方法执行序列，即方法/消息路径。与传统软件一样，方法也可能有多条内部执行路径。我们选择不在面向对象集成测试的这种细节层次上操作。MM-路径从某个方法开始，当到达某个自己不发送任何消息的方法时结束，这就是消息静止点。

定义

面向对象软件中的MM-路径是由消息连接起来的方法执行序列。

第14章曾经介绍过原子系统功能（ASF），请考虑支持给定ASF的类集合。ASF的系统级性质意味着，它最低地表示端口级事件的激励/响应对偶。由于端口事件（很有可能）与不同类的操作关联，因此在这个层次上实现不同类的协作。我们只需要对传统软件测试中的两个定义做少许修改。

定义

原子系统功能（ASF）是一种MM-路径，从输入端口事件开始，到输出端口事件结束。

ASF概念描述了面向对象软件的事件驱动性质。由于ASF常常从端口输入事件开始，到端口输出事件结束，因此ASF构成传统模型所说的激励/响应路径。这种系统级输入会触发MM-路径的方法-消息序列，这个序列有可能触发其他MM-路径，直到最终MM-路径序列以某个端口输出事件结束。当这种序列结束时，系统就是事件静止的，即系统在等待启动另一个ASF的另一个端口输入事件。就像传统软件一样，这种方法将ASF测试放在集成和系统级测试的重叠点上。

o-oCalendar的伪代码

为了看出面向对象集成测试的复杂性，下面详细讨论o-oCalendar应用程序。以下给出这

个应用程序的伪代码。消息传输情况如图18-3所示（为了减轻通路拥挤问题，使框图清晰，图中只给出消息编号）。

```
class CalendarUnit    'abstract class
    currentPos As Integer

    setCurrentPos(pCurrentPos)
        currentPos = pCurrentPos
    End    'setCurrentPos

    abstract protected boolean increment()

class testIt
    main()
        testdate = instantiate Date(testMonth, testDay, testYear)    msg1
        testdate.increment()                                         msg2
        testdate.printDate()                                         msg3
    End    'testIt

class Date
    private Day d
    private Month m
    private Year y

        Date(pMonth, pDay, pYear)
            y = instantiate Year(pYear)                              msg4
            m = instantiate Month(pMonth, y)                         msg5
            d = instantiate Day(pDay, m)                             msg6
        End    'Date constructor

        increment()
            if (NOT(d.increment()))                                  msg7
        Then
            if (NOT(m.increment()))                                  msg8
            Then
                y.increment()                                        msg9
                m.setMonth(1,y)                                      msg10
            Else
                d.setDay(1, m)                                       msg11
            EndIf
        EndIf
        End    'increment

    printDate()
        Output (m.getMonth() + "/" + d.getDay() + "/" + y.getYear())msg12, msg13, msg14
    End    'printDate

    class Day isA CalendarUnit
    private Month m

    Day(pDay, Month pMonth)
        setDay(pDay, pMonth)                                         msg15
        End    'Day constructor

    setDay(pDay, Month pMonth)
        setCurrentPos(pDay)                                          msg16
        m = pMonth
```

```
        End     'setDay

26  getDay()
27      return currentPos
        End     'getDay

28  boolean increment()
29      currentPos = currentPos + 1
30      if (currentPos <= m.getMonthSize())                    msg17
31          Then    return true
32          Else    return false
33      EndIf
        End     'increment

        class Month isA CalendarUnit
        private Year y
        private sizeIndex = <31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31>

34  Month( pcur, Year pYear)
35      setMonth(pCurrentPos, Year pyear)                      msg18
        End     'Month constructor

36  setMonth( pcur, Year pYear)
37      setCurrentPos(pcur)                                    msg19
38      y = pYear
        End     'setMonth

39  getMonth()
40      return currentPos
        End     'getMonth

41  getMonthSize()
42      if (y.isleap())                                        msg20
43          Then    sizeIndex[1] = 29
44          Else        sizeIndex[1] = 28
45      EndIf
46      return sizeIndex[currentPos -1]
        End     'getMonthSize

47  boolean increment()
48      currentPos = currentPos + 1
49      if (currentPos > 12)
50          Then    return false
51          Else    return true
52      EndIf
        End     'increment

            class Year isA CalendarUnit

53  Year( pYear)
54      setCurrentPos(pYear)                                   msg21
        End     'Year constructor

55  getYear()
56      return currentPos
        End     'getYear
```

```
boolean increment()
    currentPos = currentPos + 1
    return true
    End    'increment

boolean isleap()
    if (((currentPos MOD 4 = 0) AND NOT(currentPos MOD 400 = 0)) OR
        (currentPos MOD 400 = 0))
        Then    return true
        Else    return false
        EndIf
    End    'isleap
```

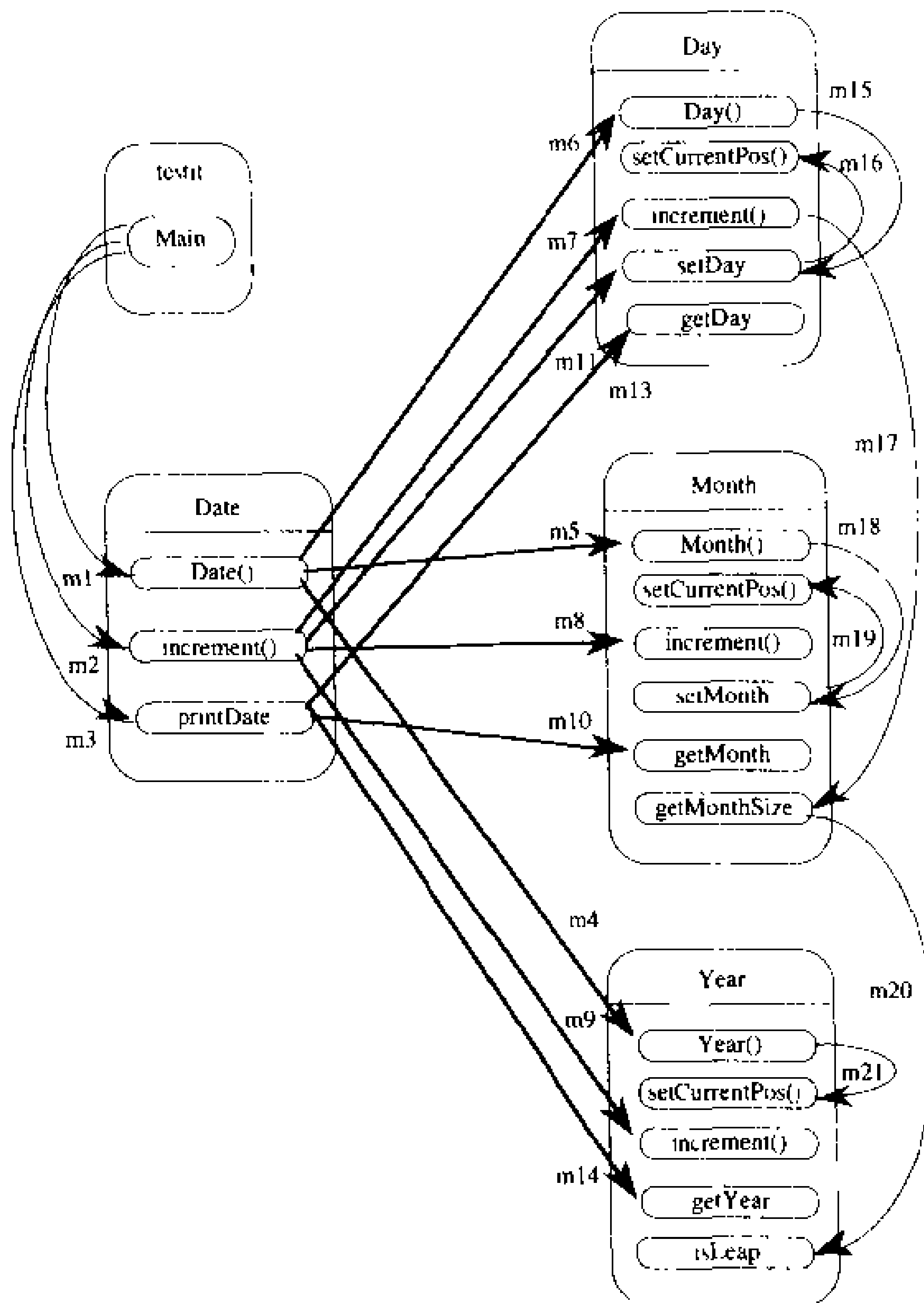


图18-3 o-oCalendar中的消息

图18-3是图18-1中协同图的更详细的视图，两者的差别是，前者显示了每个方法的源方法和目的地方法，而不只是类。采用这种方法，我们可以独立地观察面向对象集成测试，判断是以方法还是以类为单元。

以下是采用2002年1月15日实例化Date的部分MM-路径。与第13章一样，这里也直接使用伪代码中的语句和消息编号。这个MM-路径如图18-4所示。

```
testIt<1>
    msg1
Date:testdate<4, 5>
    msg4
Year.y<53, 54>
    msg21
Year.y setCurrentPos<a, b>
    (return to Year y)
    (return to Date:testdate)
Date:testdate<6>
    msg5
Month.m<34, 35>
    msg18
Month:m.setMonth<36, 37>
    msg19
Month:m.setCurrentPos<a, b>
    (return to Month:m.setMonth)
    (return to Month:m)
    (return to Date:testdate)
Date.testdate<7>
    msg6
Day:d<21, 22>
    msg15
Day:d.setDay<23, 24>
    msg16
Day:d.setCurrentPos<a, b>
    (return to Day:d.setDay)
Day:d.setDay<25)
    (return to Day:d)
    (return to Date:testdate)
```

以下是用2002年4月30日实例化的更有意思的MM-路径（请参阅图18-5）。

```
testIt<2>
    msg2
Date:testdate increment<8,9>
    msg7
Day:d.increment<28, 29>  'now Day.d.currentPos = 31
    msg17
Month:m.getMonthSize<41, 42>
    msg20
Year:y.isleap<60, 61, 63, 64>  'not a leap year
    (return to Month:m.getMonthSize)
Month:m.getMonthSize<44, 45, 46>        'returns month size = 30
    (return to Day:d.increment)
Day:d.increment<32, 33>        'returns false
    (return to Date:testdate.increment)
```

```
Date:testdate.increment<10, 11>
    msg8
Month:m.increment<47, 48, 49, 51, 52>        'returns true
    (return to Date:testdate.increment)
Date:testdate.increment<15, 16>
    msg11
Day:d.setDay<23, 24, 25>      'now dayis 1, month is 5
    (return to Date:testdate.increment)
Date:testdate.increment<17, 18>
    (return to testIt)
```

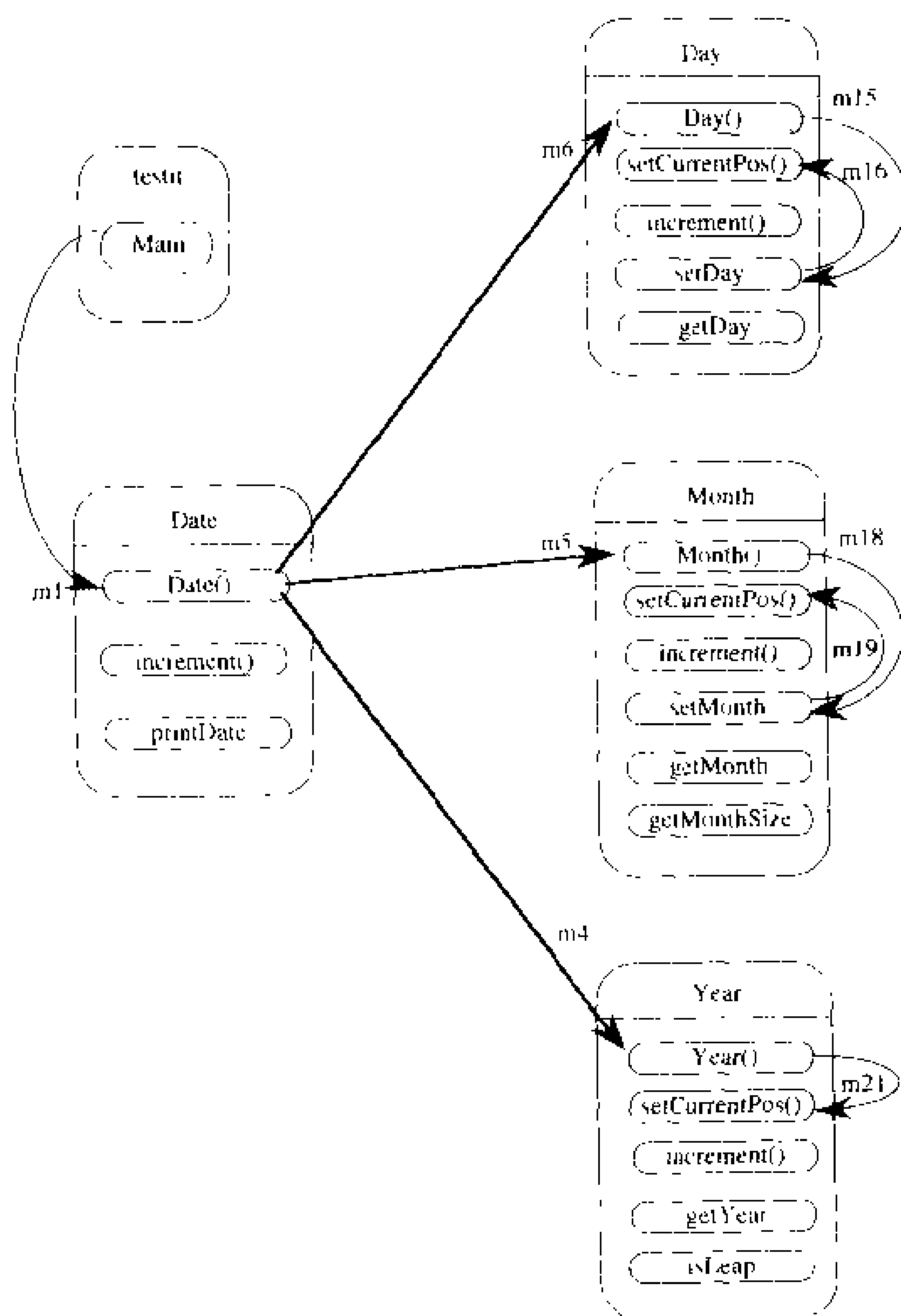


图18-4 实例化Date（2002年1月15日）的MM-路径

有向图的方式使我们能够基于MM-路径分析选择集成测试用例。首先我们要问需要多少测试用例。图18-3中有向图的圈复杂度是23（每个消息的返回边没有显示出来，但是必须

计算在内）。虽然我们肯定会找出同样数量的基本路径，但是不必这样做，因为一条MM-路径会覆盖这些路径中的很多条，而还有很多路径在逻辑上是不可行的。较低的限制是三个测试用例：从testIt伪代码中1、2和3语句开始的MM-路径。这样做可能是不充分的，因为如果选择的“容易”日期（例如2002年1月15日），就不会出现涉及isLeap和setMonth的消息。正如我们在过程代码单元级测试所看到的情况一样，最低限度同样也需要覆盖所有消息的一组MM-路径。第7章为NextDate程序定义了13个基于决策表的功能测试用例（请参阅表7-16），构成了o-oCalendar应用程序透彻的集成测试用例集合。在这一点上，面向对象集成测试的结构视图能够得到功能视图不能得到的知识。我们可以寻找MM-路径，以保证图18-3中的所有消息（边）都被遍历。

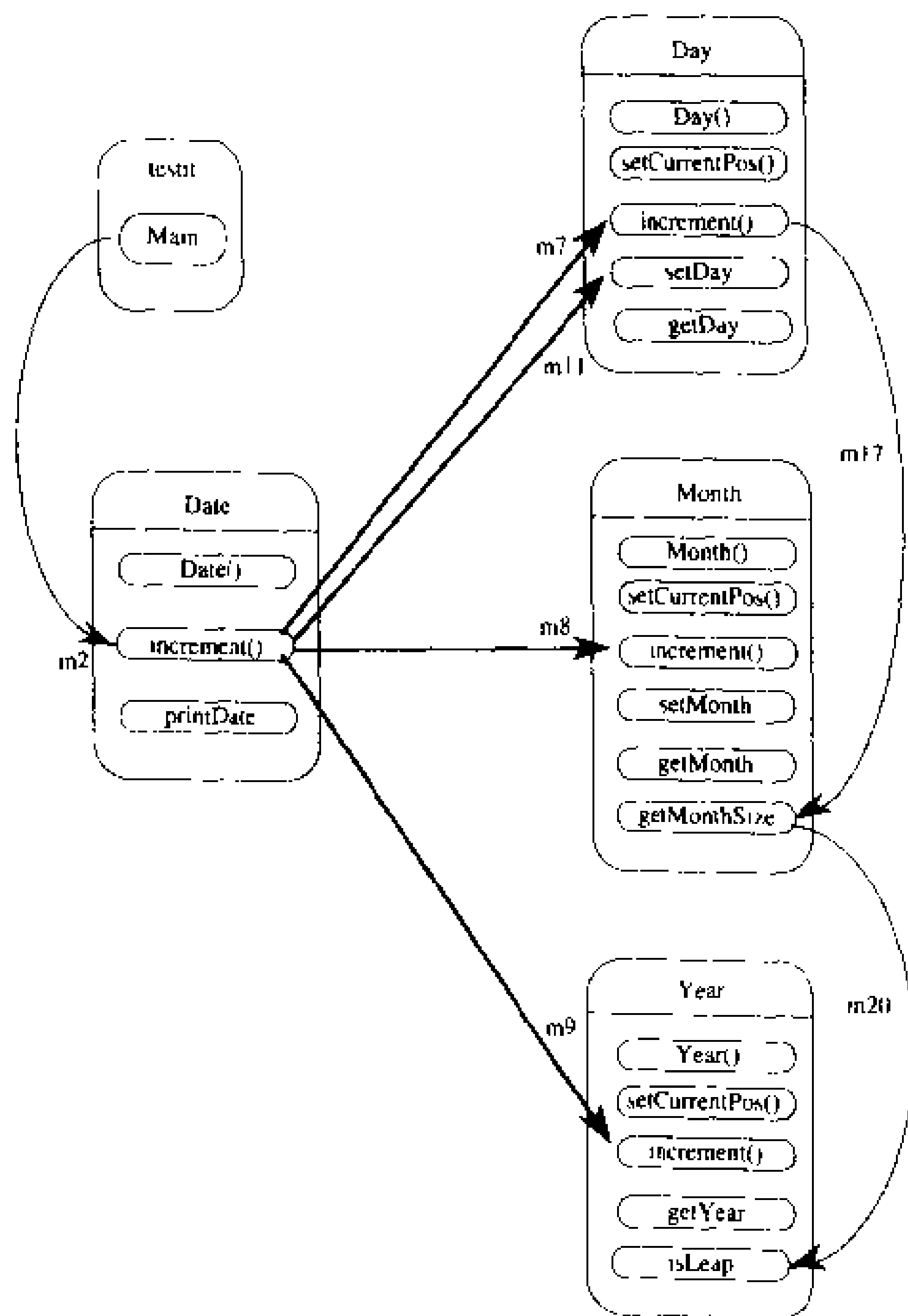


图18-5　Date.increment（2002年4月30日）的MM-路径

18.3　面向对象数据流集成测试框架

定义集成测试的MM-路径可与DD-路径类比。正如讨论过程软件时所介绍的那样、

DD-路径测试常常是不充分的，在这种情况下，采用数据流测试更合适。对于面向对象软件的集成测试也是这样，且需求更强烈。这是因为：（1）取值可以通过继承树得到；（2）数据可以在消息传递的不同阶段定义。

程序图构成过程代码的描述和分析数据流测试的基础。面向对象软件的复杂性超出有向（程序）图的表达能力。本节介绍一种描述框架，在这种框架中，可以描述和分析面向对象软件的数据流测试问题。

18.3.1 事件驱动和消息驱动的Petri网

第4章定义了事件驱动的Petri网，这里我们对其扩展以描述对象之间的消息通信。图18-6给出了在事件驱动和消息驱动Petri网（EMDPN）中使用的表示符号。

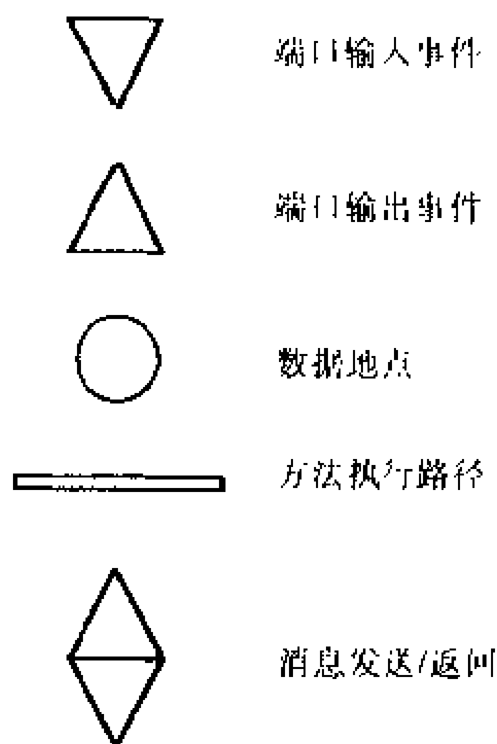


图18-6 EMDPN的符号

定义

EMDPN是一种由四部分组成的有向图（P、D、M、S、In、Out），包括四组节点：P、D、M和S，以及两个映射：In和Out。其中：

P是端口事件集合。

D是数据地点集合。

M是消息地点集合。

S是转移集合。

In是（P∪D∪M）×S的有序对偶集合。

Out是S×（P∪D∪M）的有序对偶集合。

我们保留端口输入和输出事件，因为这些事件肯定会出现在事件驱动的面向对象应用程序中。显然，我们仍然需要数据地点，把Petri网转移解释为方法执行路径。新符号用来获取

对象间消息的本质：

这些消息是发送消息对象中的方法执行路径一个输出
这些消息是接收消息对象中的方法执行路径一个输入，
返回是接收消息对象中的方法执行路径一个非常微妙的输出
返回是发送消息对象中的方法执行路径一个输入

图18-7给出的是新消息地点可以在EMDPN中出现的惟一方式。

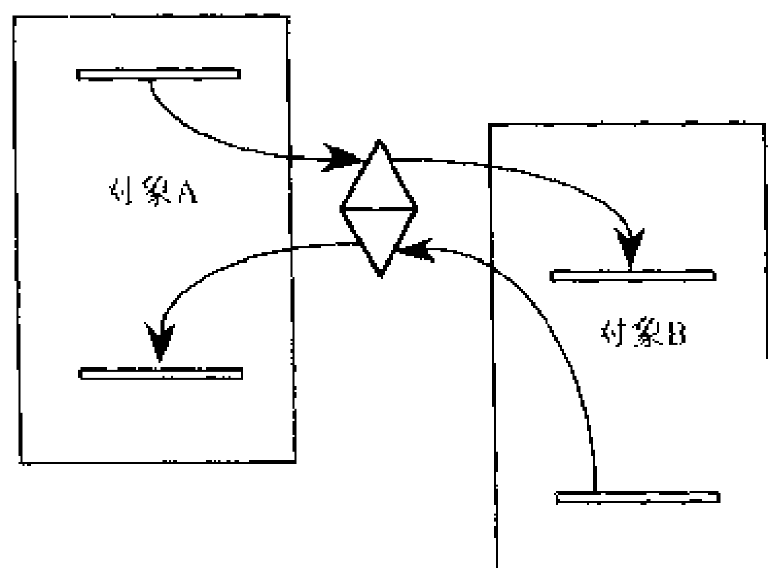


图18-7 对象之间的消息连接

由于EMDPN是一种有向图，因此其结构提供面向对象软件数据流分析的框架。前面已经介绍过，过程代码的数据流分析以定义和使用值的节点为中心。在EMDPN框架中，数据由数据地点表示，值在方法执行路径中定义和使用。数据地点可以是到方法执行路径的输入或输出，因此我们现在可以表示以与过程代码非常类似的方式定义和使用路径（定义-使用路径）。尽管有四类节点，不过仍然有这些节点之间的路径，因此可以在定义-使用路径中忽略节点的类型，只关注其连通性。

18.3.2 由继承导出的数据流

请考虑定义了数据项值的继承树，在这种树中，考虑从定义了值的数据地点开始，到树"尾"结束的链。这种链将是一种替代数据地点序列，并退化方法执行路径，方法执行路径在链中实现面向对象语言的继承机制。因此，这种框架支持多种形式的继承，包括单一继承、多重继承和有选择性的继承。描述继承的EMDPN仅由数据地点和方法执行路径组成，如图18-8所示。

18.3.3 由消息导出的数据流

图18-9中的EMDPN给出了三个对象之间的消息通信。作为定义-使用路径的一个例子，假设mep3是要由mep5转发，在mep6中修改，并最终在Use节点mep3中使用的数据项的"定

义”节点。我们可以标识这两条定义-使用路径：

du1=<mep3，msg2，mep5，d6，mep6，return(msg2)，mep4，return(msg1)，mep2>

du2=<mep6，return(msg2)，mep4，return(msg1)，mep2>

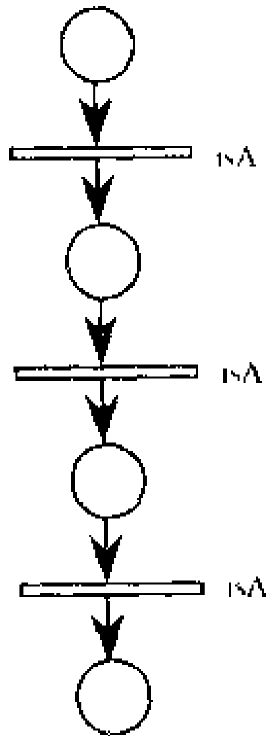


图18-8 有继承的数据流

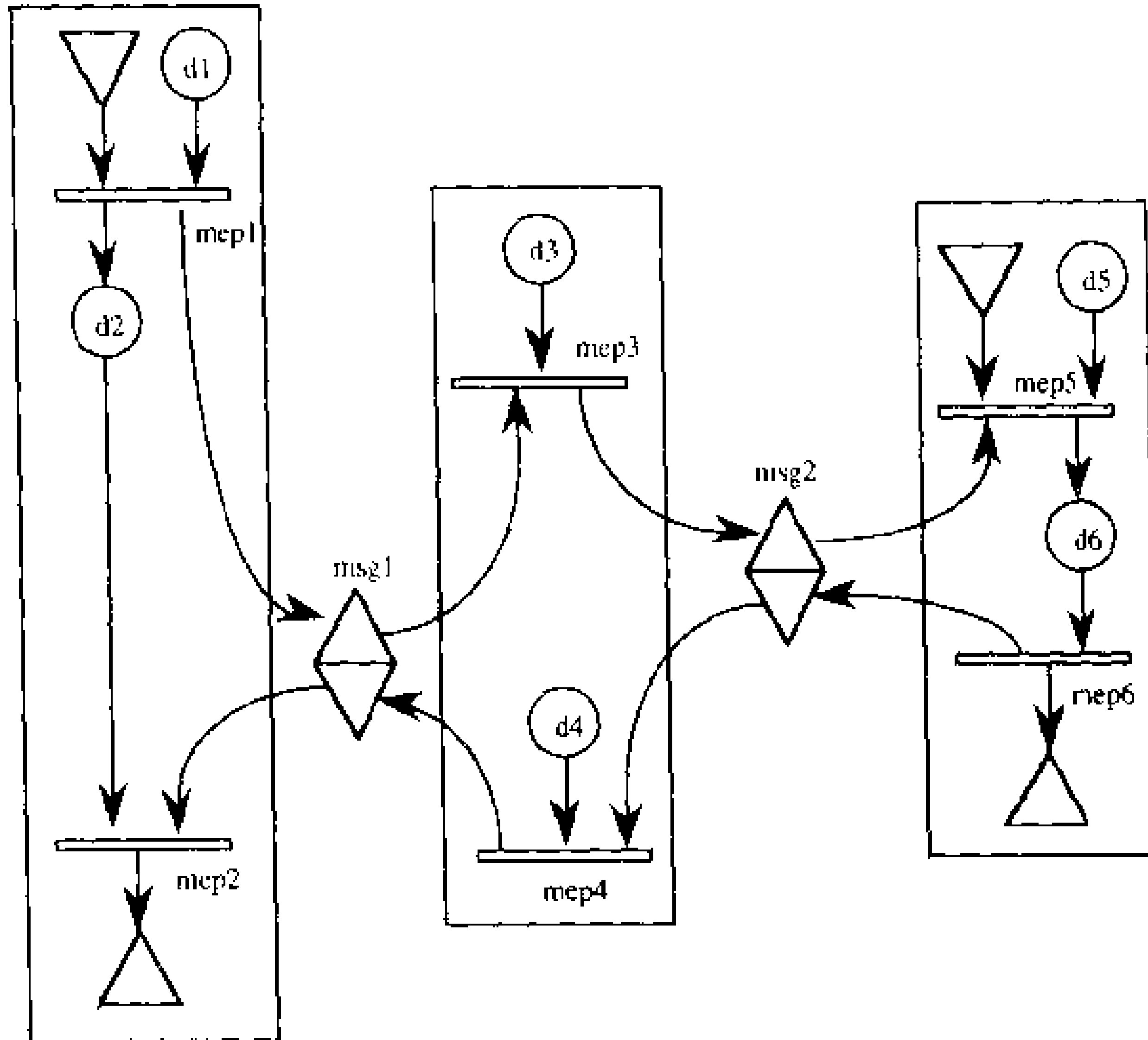


图18-9 通过消息通信的数据流

在这个例子中，du2是定义清除的，而du1不是。

18.3.4 分片

人们很希望说这种方法也支持面向对象软件的分片。由于已经介绍了基本原理，因此我们还是继续利用图论。前面已经介绍过，最有用的分片形式是可执行分片。这看起来是一种很大的扩展，如果分片不能执行，则分片只能用于定位缺陷的桌面审查方法。

18.4 练习

下面对形式做一个大的改变。我妻子辅导的六年级班曾上演一出戏剧，这是从中节选出的一段台词。这段台词可以与面向对象集成测试类比。通过这种类比，我们可以将所讨论的集成测试概念用到戏剧情节的排练中。首先把剧中的人物看做类。由于人物是通过线索沟通的，因此可以将人物台词看做给另一个人物的消息。

首先给出“源代码”（《幽灵公路收费站》（The Phantom Tollbooth）第1场第2幕节选，作者Susan Janus（根据Norton Juster小说改编））：

1. **看门狗**。词汇城，我们到了。

2. **Milo**。嘿，看门狗，你好点了吗？

3. **Tock**。就叫我Tock吧，看着点路。

4. **Milo**。不过词汇城是个什么地方？

5. **Tock**。全世界的词汇都来自那个地方。那曾经是一个很好的地方，不过自从韵律和推理离开后，就再也不是从前的词汇城了。

6. **Milo**。韵律和推理？

7. **Tock**。是两位公主。她们的两个兄弟统治着智慧大地，两位公主曾经解决自己两兄弟之间的所有争执。字典大全是词汇城的国王，而数学魔王则是数字城的国王，他们在所有事情上几乎都不能有一致的看法。甜韵律和纯推理两位公主只好解决两位国王的矛盾，而且每当发生矛盾的时候，她们总是能够使两位国王满意地回家。不过有一天，两位国王的争论结束了所有争论……

[Tock和Milo身上的灯光渐暗，词汇城国王字典大全出现在舞台的另一边。他腆着硕大的肚子，灰色的胡须一直拖到腰际，头上扣着一顶很小的王冠，长袍上写满了字母。]

8. **字典大全**。当然，我能容忍韵律和推理的决定，尽管我对未来的结果深信不疑。任何人都知道每时每刻词汇都要比数字更重要。

[数学魔王出现在字典大全的对面。他穿着长长的飘动着的袍子，戴着又高又尖的帽子，袍子上面写满了复杂的数学公式。他拄着一根长拐杖，拐杖的一端是铅笔尖，另一端是一个大橡皮头。]

9. **数学魔王**。那是你的想法，字典大全。没有数字，人们连一周的第几天都不知道。你难道没有见过日历吗？看看日历吧，字典大全，日历是靠数字计算的。

10. **字典大全** 别犯傻了 [面对观众，做领头鼓掌状]让我们听听词汇得到的掌声吧！

11. **数学魔王** [面对观众，也做领头鼓掌状]把你们手中的票投给数字吧！

12. **字典大全** A、B、C!

13. **数学魔王** 1、2、3!

14. **字典大全和数学魔王** [二人相对]安静！韵律和推理就要宣布她们的决定了

15. **韵律** 女士们、先生们、字母们、数字们，分数们、标点们，大家请注意！我们仔细考虑了词汇城国王字典大全[字典大全鞠躬示意]和数字城国王数学魔王[数学魔王举手行胜利礼]提出的问题，结论如下

16. **推理** 对于知识斗篷来说，词汇和数字具有相等价值，一个是斗篷的经线，一个是斗篷的纬线。

17. **韵律** 数沙粒并不比叫出星星的名字更重要

18. **韵律和推理** 因此，让词汇城王国和数字城王国都相安无事吧！

[掌声响起]

19. **字典大全** 嘘！我只能说嘘 嘘，呸！

20. **数学魔王** 如果这些姑娘不能使任何一方满意地解决争端，那还有什么意思呢？我认为我应该自己做出决定

21. **字典大全** 我也应该

22. **字典大全和数学魔王** [面对两位公主]从今以后，你们只能呆在空中城堡，不许再到陆地上来 [二人相对]至于你，快从这滚开！[两人大步向相反方向离开]

[这时，布景换为词汇城的集贸广场。舞台灯光照亮的是已经被遗弃了的广场]

23. **Tock** 从那以后，在这个王国中就再也没有出现韵律和推理 词汇被到处乱用，数字被胡乱计算 两个国王之间的争论分散了所有人，词汇和数字的真正价值被遗忘 这是多大的浪费！

24. **Milo** 为什么没有人营救公主，使一切都恢复正常呢？

25. **Tock** 说起来容易做起来难，空中城堡离这里非常远，而且通向那里的惟一道路由凶恶的魔鬼把守着。不过且慢，我们来了 [有一个人出现，拿着一扇大门和一个很小的公路收费站。]

26. **看门人** 哎！这是词汇城，是个快乐王国，得天独厚地座落在朦胧的丘陵中，沐浴着知识之海的和风。根据皇家命令，今天是集贸日 你们到这来做买卖吗？

27. **Milo** 对不起，你说什么？

28. **看门人** 做买卖，就是买和卖 是买还是卖？你们到这里来总是有原因的吧？

29. **Milo** 嗯，我……

30. **看门人** 快说吧，如果你们没有原因，至少总有个解释或借口吧

31. **Milo** [怯懦地]嗯，……没有

32. **看门人** [摇头]说真的，你们没有原因是不能进去的 [若有所思地]等一下 可能我还有一个旧的你们可以用 [从公路收费站中掏出一个旧皮箱翻了起来]不……不……这不行……嗯……

33. Milo。[对Tock。]他在找什么？[Tock耸了耸肩膀。]

34. 看门人。啊！很好。[掏出一个带链子的奖章。奖章上刻着的是"为什么不？"]为什么不，这差不多是所有事的很好理由……也许被用得太滥了一点，但仍在使用。给您，先生。现在我可以真正地说："词汇城欢迎您。"

最后，请做以下练习：

1. 请为戏剧中的人物开发与调用图（或协同图）类比的框图。

2. 成对集成的类比是什么？作为一种排练，技术有多大作用？请列出可能一起排练的人物对。

3. 相邻集成的类比描述是什么？作为一种排练，技术有多大作用？请列出至少两个人物邻居。

4. 能够找出对应于ASF的事物吗？

5. 戏剧导演可能会以与软件开发项目中集成测试构建相当类似的方式安排排练。请找出可能的构建。

6. 一般排练与彩排的不同，以什么方式能够与集成测试和系统测试之间的差别相对应？

7. 如果把人物看做对象，把剧情看做消息，进行类比会出现什么情况。将第1~7段作为第1场，第8~22段作为第2场，第23~34段作为第3场，请为这三场画出相应的序列图。

18.5 参考文献

Regmi, D.R., Object-Oriented Software Construction Based on State Charts, Grand Valley State University Master's project, Allendale, Michigan, 1999.

第19章

GUI 测 试

所有图形用户界面（GUI）应用程序的主要特征都是事件驱动。用户可以以任何顺序引发多个事件中的任意几个。虽然可以创建“引导”事件序列的GUI应用程序，但是很多GUI应用程序并没有这类功能。GUI应用程序为测试人员提供了一点方便：基本上不需要集成测试，单元测试一般在“按钮级”进行，也就是说，按钮具有功能，而且这些功能可以在一般单元级意义上进行测试。GUI应用程序的系统级测试的本质，是表现出应用程序的事件驱动性质。但是，统一建模语言（UML）中的大多数模型对事件驱动的系统没有什么帮助。主要的例外是行为模型，尤其是状态图及其更简单的有限状态机的情况。

19.1 货币转换程序

货币转换程序是一种事件驱动程序，强调与GUI关联的代码。采用Visual Basic构建的一个样本GUI应用程序如图19-1所示（与第2章的图一样）。

货币转换器
美元金额
等价于……
巴西
加拿大
欧共体
日本
计算
清除
退出

图19-1 货币转换器GUI

这个应用程序将美元转换为四种货币中的任何一种：巴西瑞尔、加拿大元、欧共体欧元和日元。货币选择由无线电按钮（Visual Basic选项按钮）控制，这些按钮相互排斥。当选择了一个国家时，系统会使标签句子变得完整。例如，如果点击了“加拿大”按钮，则标签“等于……”会变成“等于加拿大元”。此外，程序会在等量货币金额的输出位置旁边显示

一面加拿大国旗。选择货币之前或之后，用户输入美元金额。一旦两个任务都完成，用户可点击“计算”按钮、“清除”按钮或“退出”按钮。点击“计算”按钮会将美元金额转换为所选货币的等量金额。点击“清除”按钮可重新设置货币选择、美元金额和等量货币金额及相关的标签。点击“退出”按钮，结束该应用程序。

为了测试GUI应用程序，首先标识所有用户输入事件和所有系统输出事件（这些事件都必须是外部可视和可观察的）。货币转换程序的输入输出事件如表19-1所示。

表19-1　货币转换程序的输入与输出事件

	输入事件		输出事件
ip1	输入美元金额	op1	显示美元金额
ip2	按下国家按钮	op2	显示货币名称
ip2.1	按下巴西	op2.1	显示巴西瑞尔
ip2.2	按下加拿大	op2.2	显示加元
ip2.3	按下欧共体	op2.3	显示欧元
ip2.4	按下日本	op2.4	显示日元
ip3	按下“计算”按钮	op2.5	显示省略号
ip4	按下“清除”按钮	op3	指示所选国家
ip5	按下“退出”按钮	op3.1	指示巴西
ip6	在错误消息中按下OK	op3.2	指示加拿大
		op3.3	指示欧共体
		op3.4	指示日本
		op4	重新设置所选国家
		op4.1	重新设置巴西
		op4.2	重新设置加拿大
		op4.3	重新设置欧共体
		op4.4	重新设置日本
		op5	显示外币值
		op6	错误消息：必须选择国家
		op7	错误消息：必须输入美元金额
		op8	错误消息：必须选择国家并输入美元金额
		op9	重新设置美元金额
		op10	重新设置等价货币金额

图19-2给出了这个应用程序更完整的高层视图。状态是GUI外部可视的外观。例如，当程序启动、且没有用户进行输入时，程序就处于“空闲”状态（如图19-1所示）。当出现按下“清除”按钮事件时，程序也处于“空闲”状态。当用户输入美元金额并且还没有做其他事情时，程序处于“输入美元金额”状态。其他状态也以类似方式命名。请注意，“国家选择”状态实际上是一种“宏状态”，指四个被选国家中的一个。图19-3更详细地说明了这种情况。转移上的标注是表19-1给出的输入和输出事件缩写。但即使是这个小小的GUI程序也如此复杂。可以通过假想输入事件，例如ip2，在一定程度上降低复杂性：“按下国家按钮”。请注意，人工输入事件ip2和人工输出事件op2、op3和op4，大大简化了如图19-2所示的高层有限状态机。这样做省略了一些事件，尤其是按下“清除”和“退出”按钮事件。这些事

件可以在任何状态中发生，显示这些事件会使图很散乱。

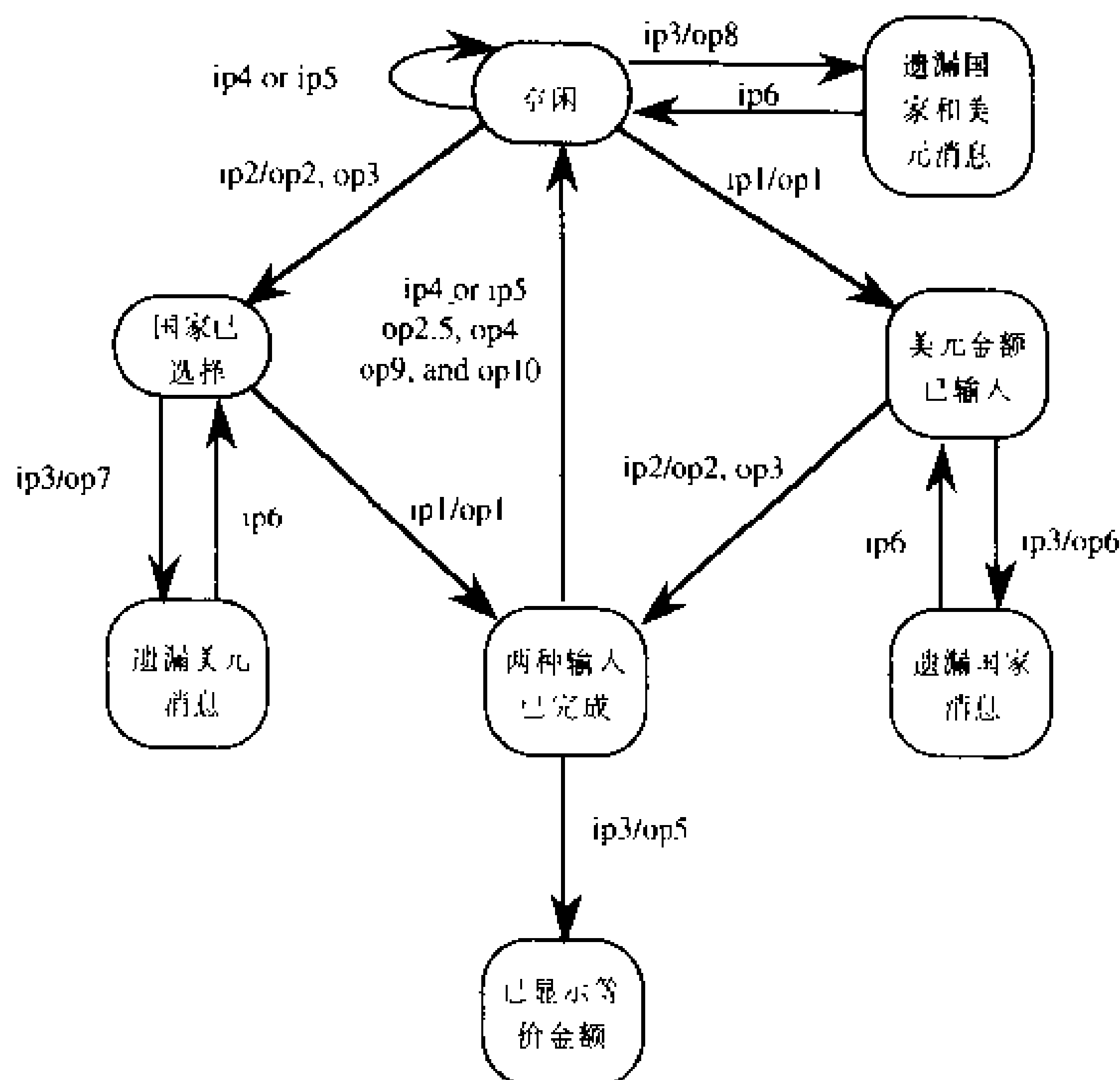


图19-2 高层有限状态机

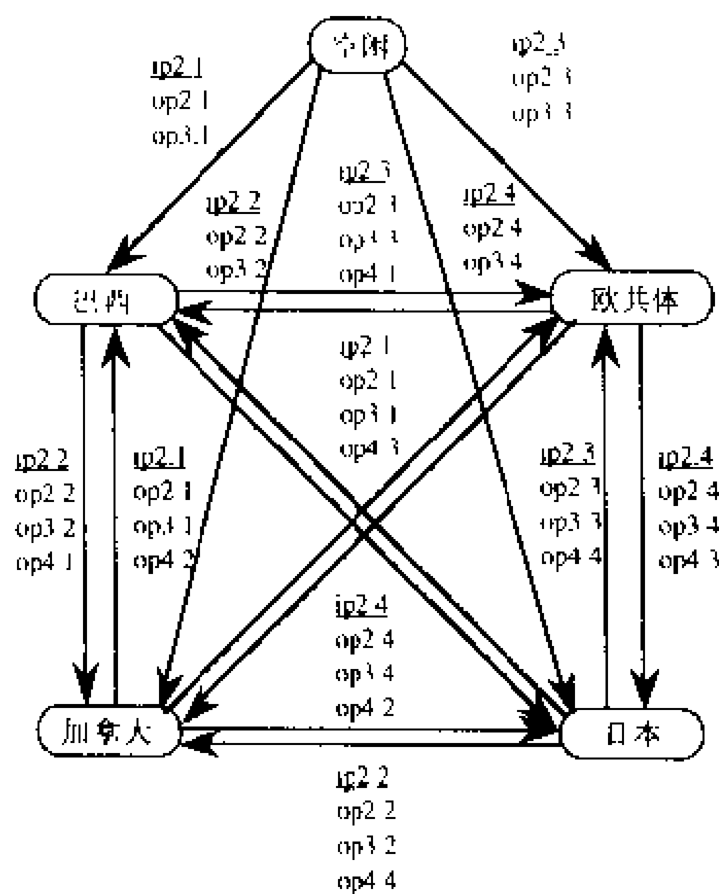


图19-3 选择国家状态详细视图

图19-3给出了“选择国家”状态内部行为的详细视图。图中维持国家按钮之间的异或关系。图19-3中更详细的视图用相应的多个实际事件替代了人工输入和输出事件。但是图19-3是不完整的，巴西和日本之间、加拿大和欧共体之间的转移标注没有给出。此外，图19-2和图19-3都没有显示在每个状态会发生的所有事件，尤其是按下“清除”或“退出”按钮事件。

熟悉“状态图”表达法的读者会熟悉将这种重复复杂性由表示法表示的很好方法。图19-4更完整地表达了图19-2的高层视图。请注意，从“执行”状态到“存储”状态的转移，显示的是用户输入ip6以退出应用程序。操作系统“结束应用程序”事件可以在执行状态内部的任何状态中出现。类似地，“清除”输入事件（ip4）可以出现在未命名状态内部的任何状态中。

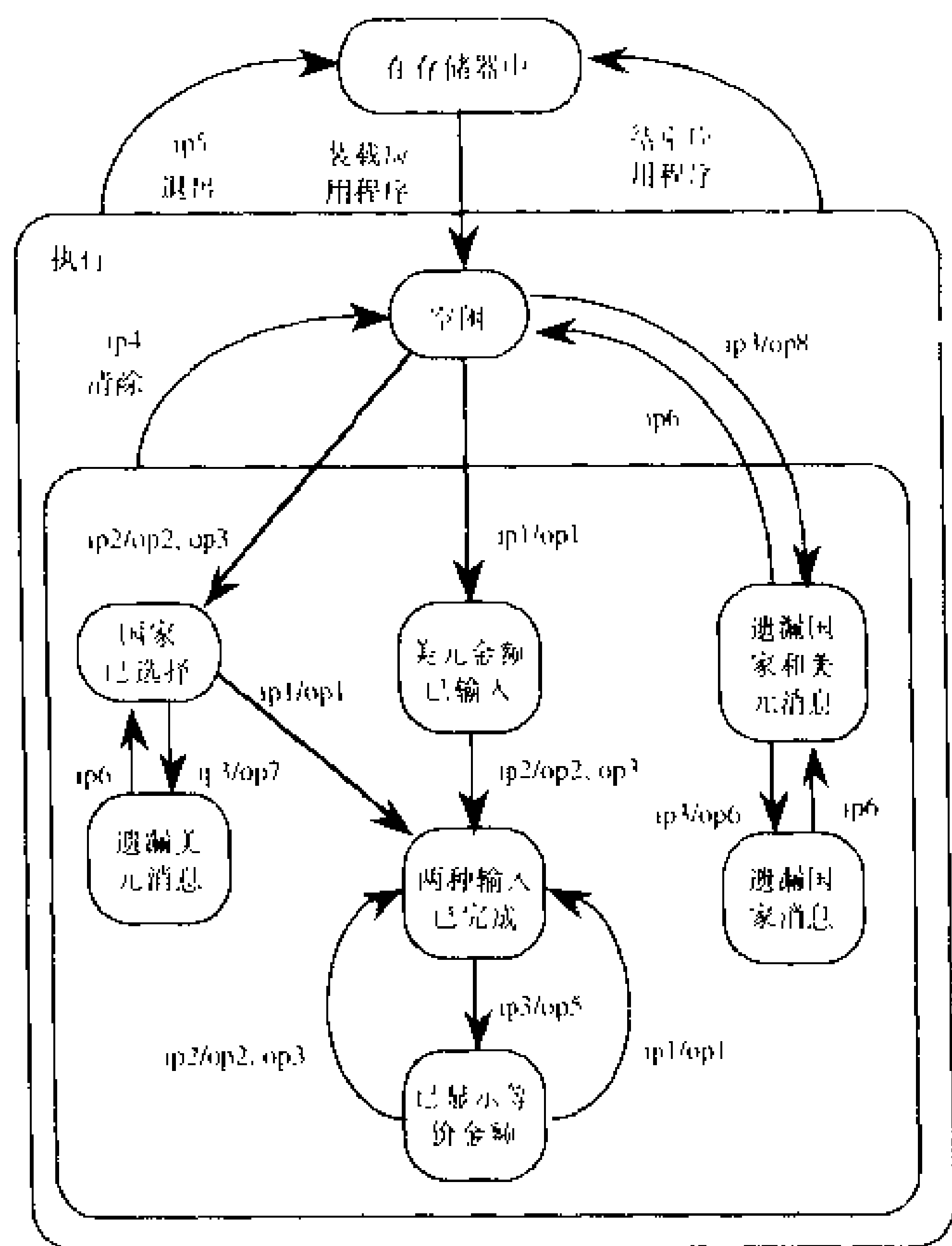


图19-4　货币转换器应用程序的状态图

处理被精简事件的一种方法，是采用单个状态层图显示对每种可能的输入事件的响应。采用事件表也可以很好地做到这一点。表19-2给出的是通过这种方法对“完成两种输入”状态的处理。

表19-2 "完成两种输入"状态的事件表

输入事件		输出事件	
ip1	输入美元金额	op1	显示美元金额
ip2.1	按下巴西	op2.1	显示巴西瑞尔
		op3.1	指示巴西
ip2.2	按下加拿大	op2.2	显示加元
		op3.2	指示加拿大
ip2.3	按下欧共体	op2.3	显示欧元
		op3.3	指示欧共体
ip2.4	按下日本	op2.4	显示日元
		op3.4	指示日本
ip3	按下"计算"按钮	op5	显示外币值
ip4	按下"清除"按钮	op2.5	显示省略号
		op4	重新设置所选国家
		op9	重新设置美元金额
		op10	重新设置等价货币金额
ip5	按下"退出"按钮		应用程序结束
ip6	在错误消息中按下OK		忽略：这个状态中的错误消息不可视！

19.2 货币转换程序的单元测试

功能按钮（"计算"、"清除"和"退出"）拥有用户提供的代码，因此是执行单元级测试的敏感地区。有些输出事件指示应该在按下"计算"事件时执行的功能，例如当没有给出输入时的错误消息。"计算"按钮的单元测试还应该考虑无效美元金额输入，例如非数字输入、负数输入和非常大的输入。在这一层既要进行功能性测试，也要进行结构性测试，本书前面对传统软件的讨论（第二部分）仍然适用。

执行单元级测试的最佳方法还没有定论。一种可能是通过将提供输入数据值特殊编码驱动器运行测试用例，并对照预期取值检查输出值。第二种方法是将GUI用做测试床。这看起来像是系统级单元测试，表面看是矛盾的，不过是可行的。说它是系统级，是因为测试用例输入要通过系统级用户输入事件提供，并且测试结果要根据系统级输出事件比较。例如，假设计算是正确的，但是失效出现在输出软件中。另一个问题是获取测试执行结果比较困难。其他问题还包括：

确实有没有经过计划的随意测试倾向

有更大的用户输入和观察错误潜在可能性

重复执行测试用例很费时间。

总的来看，采用测试驱动器的单元测试还是比较理想的。

其他更微妙的单元级测试与美元金额文本框有关，而文本框又是与具体语言有关的。例

如，如果是以Visual Basic实现，就基本不需要对文本框做单元测试。如果是真正面向对象的实现，则需要检验键盘处理器观察到的输入，是否正确地在GUI上显示出来，并且是否正确地存储在对象的属性中。

19.3 货币转换程序的集成测试

集成测试是否适用于这个例子，取决于这个例子是怎样实现的。计算按钮有三种主要选择，第一种选择关注一个地点的所有逻辑，并直接将选项按钮的状态用做条件测试中的条件，例如以下的伪代码。

```
procedure Compute ( USDollarAmount, EquivCurrencyAmount)
dim brazilRate, canadaRate, euroRate, japanRate, USDollarAmount As Single
If (optionBrazil)
    Then EquivCurrencyAmount = brazilRate * USDollarAmount
    Else
        If (optionCanada)
            Then EquivCurrencyAmount = canadaRate * USDollarAmount
            Else
                If (optionEuropeanUnion)
                    Then EquivCurrencyAmount = euroRate * USDollarAmount
                    Else
                        If (optionJapan)
                            Then EquivCurrencyAmount = japanRate * USDollarAmount
                            Else Output ("No country selected")
                        EndIF
                EndIF
        EndIF
EndIF
End procedure Compute
```

由于这些都会在单元级上彻底测试，因此没有什么必要进行集成测试。

第二种，也是要面向对象的选择，是将方法放到每个通过发送货币转换比率值响应点击事件的选项按钮对象中。（如何设置转换比率是另一个问题，目前我们假设转换比率在对象被实例化时已经定义。）经过实例化对象的伪代码，以及经过修改后的命令按钮过程如下所示：

```
object optionBrazil (USDollarExchangeRate)
    private procedure senseClick
        commandCompute( USDollarExchangeRate)
    End senseClick

object optionCanada( USDollarExchangeRate)
    private procedure senseClick
        commandCompute( USDollarExchangeRate)
    End senseClick

object optionEuropeanUnion (USDollarExchangeRate)
    private procedure senseClick
        commandCompute( USDollarExchangeRate)
```

```
    End senseClick

object optionJapan (USDollarExchangeRate)
    private procedure senseClick
        commandCompute( USDollarExchangeRate)
    End senseClick

procedure Compute ( exchangeRate, USDollarAmount)
dim exchangeRate. USDollarAmount As Single
    EquivCurrencyAmount = exchangeRate * USDollarAmount
End procedure Compute
```

请注意，作为单元，要测试的内容非常少（这就是为什么将方法选择为单元要取决于尺寸的原因。）感兴趣的所有内容都在集成级，并且在集成级，我们要考虑两个问题：选项/无线电按钮发送的是正确的转换比率值吗？等价金额计算结果正确吗？

第三种实现是Visual Basic风格。Visual Basic非常普及，值得在这里考虑。Visual Basic实现的伪代码包括转换比率全局（实际上是表单级）变量。每个选项按钮的事件过程为转换比率变量赋预先定义的值，其结果非常类似纯面向对象版本。单元测试可以由读取选项按钮事件过程的简单代码替代，并且commandCompute过程的测试也很简单。

```
Public exchangeRate As Single

Private Sub optBrazil_Click()
    exchangeRate = 1.56
End Sub

Private Sub optCanada_Click()
    exchangeRate = 1.35
End Sub

Private Sub optEuropeanUnion_Click()
    exchangeRate = 0.93
End Sub

Private Sub optJapan_Click()
    exchangeRate = 2.04
End Sub

Private Sub cmdCompute ()
    EquivCurrencyAmount = exchangeRate * Val(txtUSDollarAmount.text)
End Sub
```

请注意，在所有三个版本中，都不大需要集成测试。对于较小的GUI应用程序往往不需要集成测试，而所谓“较小”的一种比较好的界定方法，是由一个人实现的应用程序。

19.4 货币转换程序的系统测试

前面已经介绍过，至少对于较小的GUI应用程序来说，不太需要单元测试和集成测试。

这样负担就全部转移到系统测试上。由于GUI应用程序是事件驱动的，因此可以使用事件驱动的Petri网（EDPN）描述要测试的线索。表19-3列出了货币转换GUI的端口输入和输出事件，几乎与表19-1中的事件相同。表19-4列出了原子系统功能（AFS）和需要建立GUI的EDPN的数据地点。

表19-3　端口输入与输出事件

	输入事件		输出事件
p1	输入美元金额	p13	显示日元
p2	按下巴西	p14	显示省略号
p3	按下加拿大	p15	指示巴西
p4	按下欧共体	p16	指示加拿大
p5	按下日本	p17	指示欧共体
p6	按下“计算”按钮	p18	指示日本
p7	按下“清除”按钮	p19	重新设置加拿大、欧共体和日本
p8	按下“退出”按钮	p20	重新设置巴西、欧共体和日本
p9	显示美元金额	p21	重新设置巴西、加拿大和日本
p10	显示巴西瑞尔	p22	重新设置巴西、加拿大和欧共体
p11	显示加元	p23	重新设置美元金额
p12	显示欧元	p24	重新设置等价外币值
		p25	结束应用程序

表19-4　原子系统功能与数据地点

	原子系统功能		感知点击计算按钮
s1	存储美元金额	s6	感知点击“清除”按钮
s2	感知点击巴西	s7	感知点击“退出”按钮
s3	感知点击加拿大	s8	数据地点
s4	感知点击欧共体	d1	已输入美元金额
s5	感知点击日本	d2	国家已选择

表19-4稍做修改：数据地点d2叫做“国家已选择”。这比每个国家都采用一个数据地点更精确，但是却使以后画出框图变得困难。正如前面所介绍过的，表19-4代表一种可行的实现。

构建货币转换GUI的EDPN描述的下一步，是为个体原子系统功能开发EDPN，如图19-5所示。第14章已经介绍过，系统级线索是通过将原子系统功能合成为序列来构建的。图19-6给出了一个这种线索。

我们终于可以描述货币转换GUI的各种系统级测试用例集合了。最低级就是直接执行所有原子系统功能。这有一点人为因素，因为所有原子系统功能都有系统级输出不可见的数据输出。更糟糕的是，总是存在原子系统功能没有端口输出的可能性，例如s1：存储美元金额。在系统测试级，我们不能判别金额是否被正确存储，尽管我们可以查看屏幕，知道所输入的金额是正确的。

下一级系统测试是执行合适的线索集合。问题是什么线索才是“合适的”。以下是一些容易出现的情况，执行一个线索集合，使得：

使用所有原子系统功能。
使用所有端口输入。
使用所有端口输出。

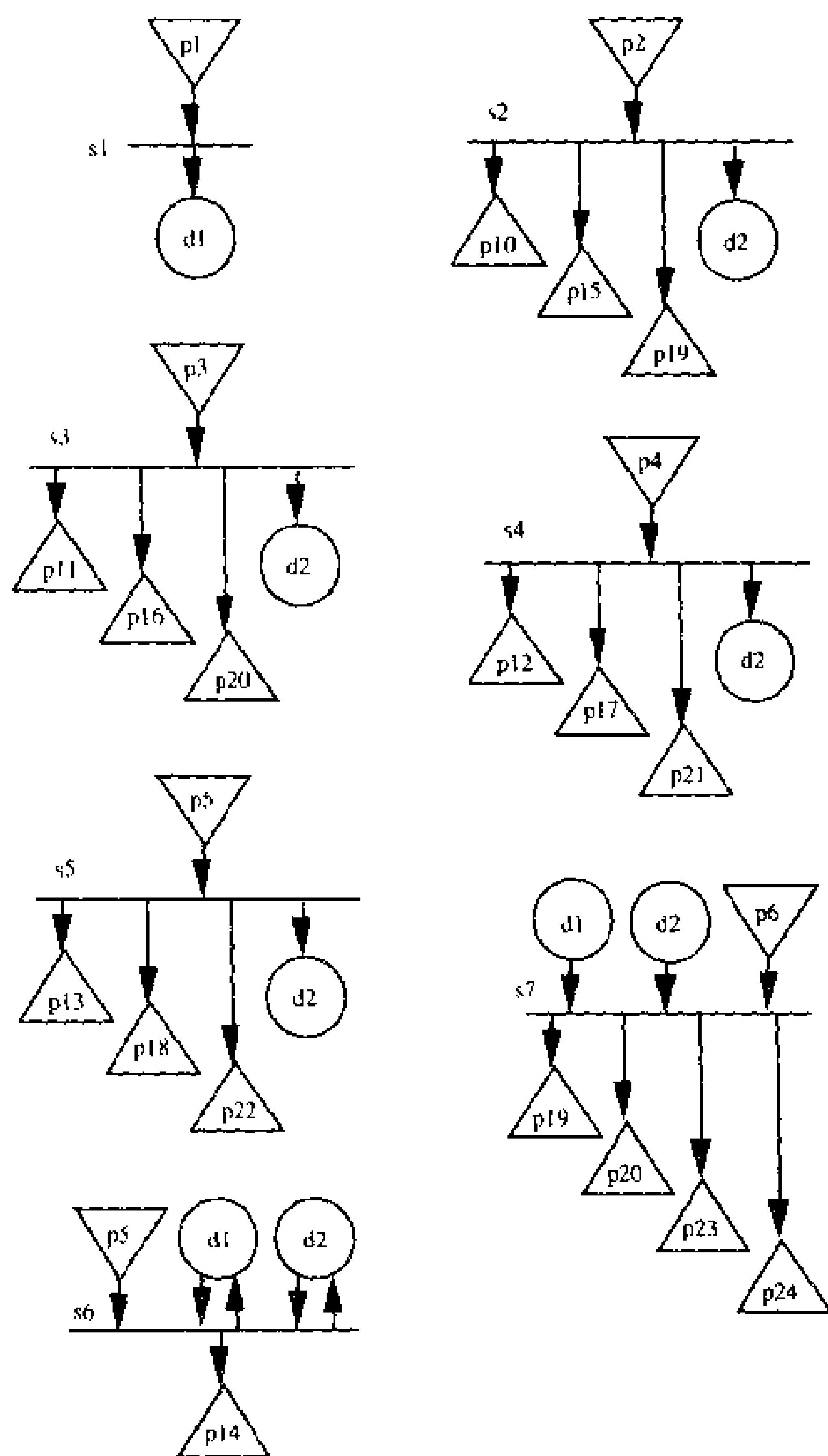


图19-5 原子系统功能的EDPN

除此之外，我们还可以研究如图19-7所示的原子系统功能有向图。这里给出的只是有向图的一部分，显示的是主线行为。线索<s1，s2，s6，s7>（如图19-6所示）是16条路径之一，半数路径都以点击“清除”按钮结束，另外一半路径以点击“退出”按钮结束。考虑线索集合T{T1，T2，T3，T4}，其原子系统功能序列如下所示：

T1 = <s1，s4，s6，s7>

T2 = <s1，s2，s6，s7>

T3 = <s3，s1，s6，s8>

T4 = <s5，s1，s7，s8>

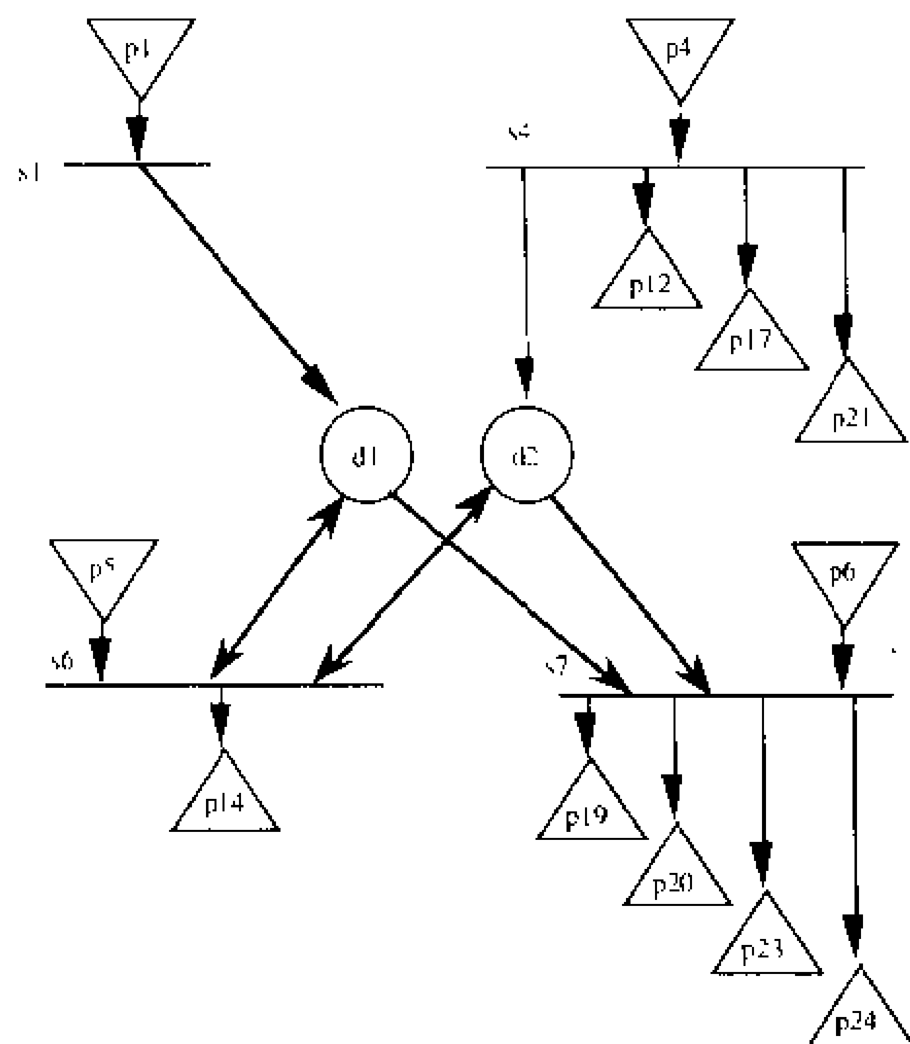


图19-6　四个原子系统功能的EDPN合成

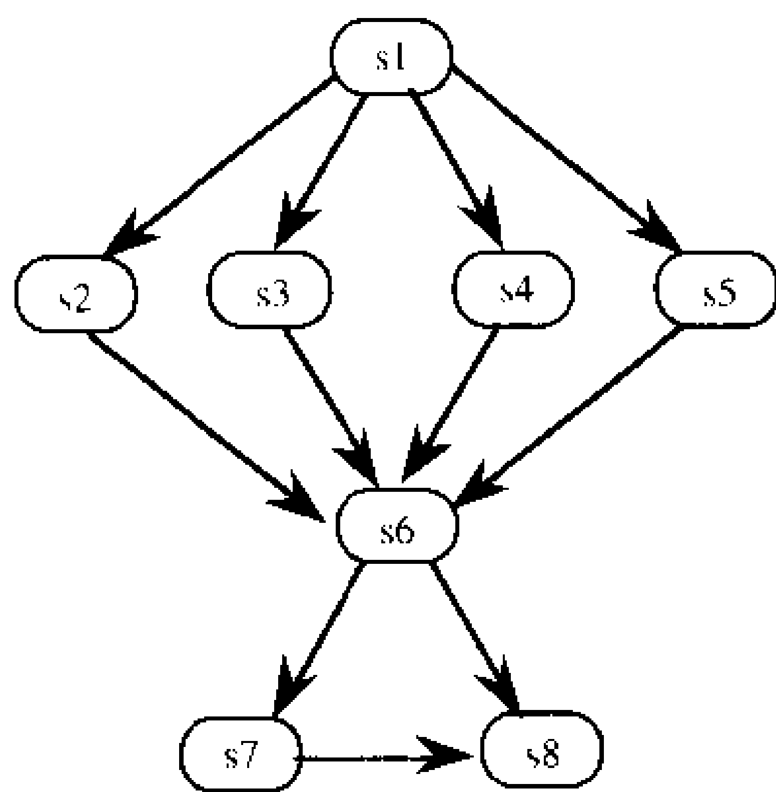


图19-7　原子系统功能的有向图

集合T具有以下“覆盖”：

所有原子系统功能；
所有端口输入；
所有端口输出。

因此，集合T构成货币转换GUI的合理最低系统测试，通过研究某些下一级用户行为，还可以更深入。线索T5 = <s1，s2，s6，s3，s6，s4，s6，s5，s6，s7，s8>是一个好例子，用户将美元金额转换为四种外币的每一种，然后“清除”屏幕并“退出”。我们还可以研究很多异常用户行为序列（行为序列是异常的，不是说用户异常！），例如T6 = <s1，s2，s3，s4，s5，s6，s7，s8>。在T6中，用户在要转换的外币上拿不定主意。我们还可以研究几乎有些傻的线索，例如T7 = <s1，s2，s3，s2，s3，s2，s3，s2，s3，s8>，用户不断在两个国家之间转换，然后退出。

像T7这样的线索可以无限长，因此在这个GUI中会出现无限多种可能的线索。采用完成图19-7部分有向图的关联矩阵可以更抽象地研究这个问题：

	s1	s2	s3	s4	s5	s6	s7	s8
s1	1	1	1	1	1	1	1	1
s2	1	0	1	1	1	1	1	1
s3	1	1	0	1	1	1	1	1
s4	1	1	1	0	1	1	1	1
s5	1	1	1	1	0	1	1	1
s6	1	1	1	1	1	1	1	1
s7	1	1	1	1	1	1	1	1
s8	0	0	0	0	0	0	0	0

19.5 练习

1. 如果仔细研究图19-5，就会发现原子系统功能s8（点击“退出”按钮）被遗漏。请画出s8的EDPN。需要什么输入？

2. GUI设计的部分艺术是防止用户输入错误。而事件驱动的应用程序更容易引入输入错误，因为事件可以以任何顺序出现。正如所给出的伪代码定义所示，用户可以输入美元金额，然后没有选择国家就点击计算按钮。类似地，用户也可以选择国家，然后没有输入美元金额就点击计算按钮。在面向对象的应用程序中，可以通过很小心地实例化对象来控制这类情况出现问题。请修改GUI类的伪代码来避免出现这两种错误

第20章

面向对象的系统测试

系统测试是（或应该是）独立于系统实现的。系统测试人员不需要真正知道实现采用的是过程代码还是面向对象的代码。第14章已经介绍过，系统测试的基本要素是端口输入和端口输出，并且介绍了任何将系统级线索表示为事件驱动的Petri网（EDPN）。问题是如何标识要被用做测试用例的线索。第14章将使用需求规格说明模型，特别是行为模型，用做线索测试用例标识的基础，还讨论了如何通过底层行为模型确定的伪结构覆盖指标。本章集中针对的是面向对象，要继承传统软件系统测试的很多思想。本章惟一真正的差别是，假设系统已经通过统一建模语言（UML）定义和细化。这样，本章的重点是通过标准UML模型找出系统级线索测试用例。

20.1 货币转换器的UML描述

本章使用货币转换器应用程序作为系统测试的一个例子。由于对象管理组（Object Management Group）的统一建模语言现在已经被广泛接受，因此我们使用Larman（1998）风格的相当完整的UML描述。本章使用的术语和UML内容基本上遵循Larman UML风格。为了扩展基本测试用例，增加了前提和后果。

20.1.1 问题陈述

货币转换器应用程序将美元转换为四种货币中的任何一种：巴西瑞尔、加拿大元、欧共体欧元和日元。用户可以修改输入，并多次进行货币转换。

20.1.2 系统功能

在有时叫做项目启动的第一步中，客户/用户要以非常一般的方式描述应用程序，形式可能是“用户使用案例”，而用户使用案例就是用例的雏形。通过这些描述，可标识出三种系统功能：显式功能、隐藏功能和装饰功能。显式功能是明显的功能；隐藏功能可能难以马上发现；装饰功能是常常出现的“锦上添花”。表20-1给出了货币转换器应用程序的系统功能。

表20-1　货币转换器应用程序的系统功能

引用编号	功　能	类　别
R1	启动应用程序	显式
R2	结束应用程序	显式
R3	输入美元金额	显式
R4	选择国家	显式
R5	执行转换计算	显式
R6	清除用户输入和程序输出	显式
R7	维护国家之间的异或关系	隐藏
R8	显示国旗图像	装饰

20.1.3　表示层

这里用得上一句老话，一张图包含的信息抵得上数千个文字包含的信息。Larman方法的第三步是画出用户界面草图。本例的用户界面如图20-1所示。用户界面的大部分信息用来支持客户走查，说明所标识的系统功能都可以由用户界面支持。

货币转换器

美元金额

等价于……

巴西

加拿大

欧共体

日本

计算

清除

退出

图20-1　用户界面草图

20.1.4　高层用例

用例开发从很高层的视图开始。请注意，随着用例开发层次的不断精化，早期的大部分信息仍然要保留。为了方便起见，可规定一种不同层次用例通用的简短的结构化命名规则。例如在这个例子中，HLUC表示高层用例（不用缩写会出现什么情况呢？）。高层用例只提供很少的细节，但是对于测试用例标识已经足够。高层用例的要点是获得在待建系统中要发生情况的很窄的描述。

HLUC 1	启动应用程序
参与者	用户
类型	基本功能
描述	用户在Windows中启动货币转换应用程序

HLUC 2	结束应用程序
参与者	用户
类型	基本功能
描述	用户在Windows中结束货币转换应用程序

HLUC 3	转换美元
参与者	用户
类型	基本功能
描述	用户输入美元金额并选择国家，应用程序计算并显示所选国家货币的等价金额

HLUC 4	修改输入
参与者	用户
类型	二级功能
描述	用户重新设置输入，开始一次新的事务处理

20.1.5 基本用例

基本用例在高层用例中增加"参与者"和"系统"事件，UML中的参与者是系统级输入的源（即端口输入事件）。参与者可以是人员、设备、相邻系统，或诸如时间这样的抽象事物。参与者行动和系统响应（端口输出事件）的编号，表示出其大致时间顺序。例如在EUC3中，观察员不能看出系统响应4和5的顺序，这两个响应看起来像是同时做出的。

EUC 1	启动应用程序	
参与者	用户	
类型	基本功能	
描述	用户在Windows中启动货币转换应用程序	
序列	参与者行动：	系统响应：
	1. 用户通过Run…命令或双击应用程序图标启动应用程序	2. 货币转换应用程序GUI显示在监视器上，并准备接收用户输入

EUC 2	结束应用程序	
参与者	用户	
类型	基本功能	
描述	用户在Windows中结束货币转换应用程序	
序列	参与者行动：	系统响应：
	1. 用户通过"退出"按钮或关闭窗口结束应用程序	2. 货币转换应用程序GUI从监视器上消失

（续）

EUC 3	转换美元	
参与者	用户	
类型	基本功能	
描述	用户输入美元金额并选择国家，应用程序计算并显示所选国家货币的等价金额	
序列	参与者行动：	系统响应：
	1. 用户通过键盘输入一个美元金额	2. 在GUI上显示美元金额
	3. 用户选择一个国家	4. 显示指定国家的货币名称
		5. 显示该国家的国旗
	6. 用户请求转换计算	7. 显示等价货币值

EUC 4	修改输入	
参与者	用户	
类型	二级功能	
描述	用户重新设置输入，开始一次新的事务处理	
序列	参与者行动：	系统响应：
	1. 用户通过键盘输入一个美元金额	2. 在GUI上显示美元金额
	3. 用户选择一个国家	4. 显示指定国家的货币名称
		5. 显示该国家的国旗
	6. 用户取消输入	7. 取消指定国家的货币名称
		8. 不再显示该国家的国旗

20.1.6 详细GUI定义

一旦标识出一组基本用例之后，就可以通过设计级细节充实图形用户界面（GUI）。这里我们采用Visual Basic实现货币转换器，并遵循Visual Basic控件的命名规则建议（Zak，2001），如图20-2所示。（对于不熟悉Visual Basic的读者，这个设计使用了四种控件：用于输入的文本框、用于输出的标签、用于指示选择的选项按钮，以及用于控制应用程序执行的命令按钮。）这些控件在扩展基本用例中会被引用。

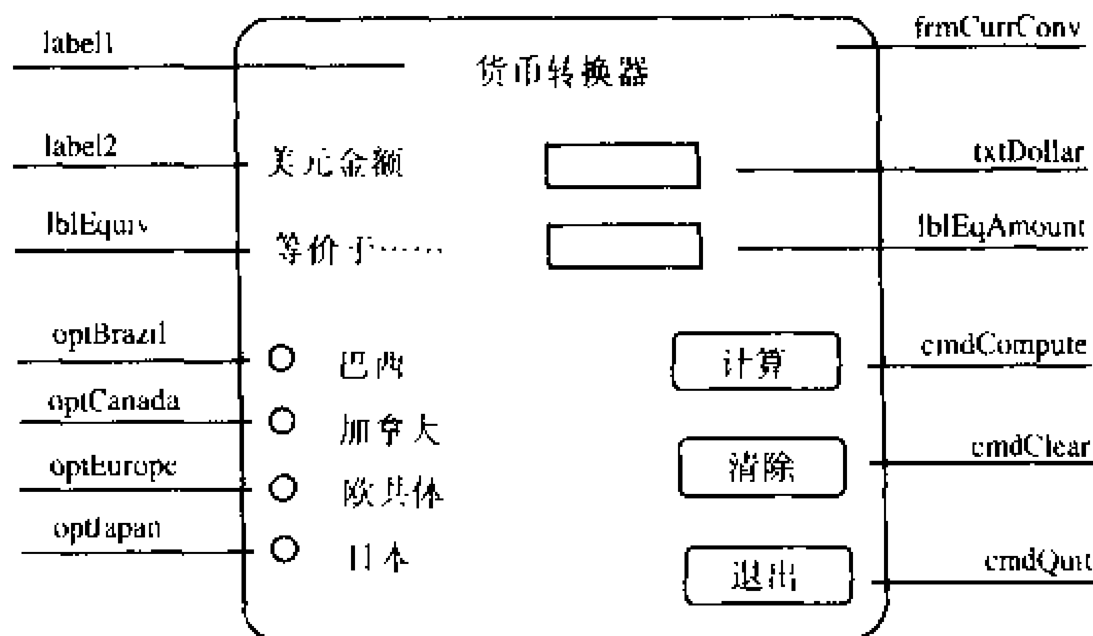


图20-2 Visual Basic控件

20.1.7　扩展基本用例

扩展基本用例是高层用例的倒数第二层细化。这里要增加前提和后果信息（不是Larman风格的一部分）、有关替代事件序列信息，以及与过程早期标识的系统功能的交叉引用。另一种扩展，是标识出更多用例并在这里添加。这是所有规格说明和设计过程的正常部分；更详细的视图提供更细致的认识。请注意，这时没有进行跨用例层次的编号跟踪。

前提和后果提供一些补充说明。我们只对直接适用于所定义的扩展基本用例的条件感兴趣。总是可以增加诸如“加电”、“计算机在Windows下运行”这样的前提。第15章已经介绍过，线索（在采用EDPN表示时）通过数据地点交互。如果对应于扩展基本用例的线索被描述为EDPN，建议记录有可能响应数据地点的前提和后果。

EEUC 1	启动应用程序	
参与者	用户	
前提	货币转换应用程序在存储器中	
类型	基本功能	
描述	用户在Windows中启动货币转换应用程序	
序列	参与者行动：	系统响应：
	1. 用户双击应用程序图标启动应用程序	2. frmCurrConv显示屏幕上
替代序列	用户采用Windows的Run…命令打开货币转换应用程序	
交叉引用	R1	
后果	txtDollar得到焦点	

EEUC 2	结束应用程序	
参与者	用户	
前提	frmCurrConv处于运行模式	
类型	基本功能	
描述	用户在Windows中结束货币转换应用程序	
序列	参与者行动：	系统响应：
	1. 用户点击cmdQuit	2. frmCurrConv被卸载
替代序列	用户关闭frmCurrConv窗口	
交叉引用	R2	
后果	货币转换应用程序在存储器中	

EEUC 3	正常使用（首先输入美元金额）
参与者	用户
前提	txtDollar得到焦点
类型	基本功能
描述	用户输入美元金额并选择国家，应用程序计算并显示所选国家货币的等价金额

（续）

序列	参与者行动：	系统响应：
	1. 用户通过键盘输入一个美元金额	2. 美元金额出现在txtDollar中
	3. 用户点击一个国家按钮	4. 指定国家的货币名称出现在lblEquiv中
	5. 用户点击cmdCompute按钮	6. 计算出的等价金额出现在lblEqAmount中
替代序列	行动1和3顺序可以颠倒，相应的响应4和6顺序也可以颠倒	
交叉引用	R3、R4、R5	
后果	cmdClear得到焦点	

EEUC 4	重复转换，相同国家	
参与者	用户	
前提	txtDollar得到焦点	
类型	基本功能	
描述	用户输入美元金额并选择国家，应用程序计算并显示所选国家货币的等价金额	
序列	参与者行动：	系统响应：
	1. 用户通过键盘输入一个美元金额	2. 美元金额出现在txtDollar中
	3. 用户点击一个国家按钮	4. 国家的货币名称出现在lblEquiv中
	5. 用户点击cmdCompute按钮	6. 计算出的等价金额出现在lblEqAmount中
	7. 用户点击txtDollar	8. txtDollar得到焦点
	9. 用户通过键盘输入另一个不同的美元金额	10. 美元金额出现在txtDollar中
	11. 用户点击cmdCompute按钮	12. 计算出的等价金额出现在lblEqAmount中
替代序列	行动1和3顺序可以颠倒，相应的响应4和6顺序也可以颠倒；行动7、9和11可以无限次地重复，相应的响应8、10和12也可以随之重复	
交叉引用	R3、R4、R5	
后果	cmdClear得到焦点	

EEUC 5	重复转换，相同美元金额	
参与者	用户	
前提	txtDollar得到焦点	
类型	基本功能	
描述	用户输入美元金额并选择国家，应用程序计算并显示所选国家货币的等价金额	
序列	参与者行动：	系统响应：
	1. 用户通过键盘输入一个美元金额	2. 美元金额出现在txtDollar中
	3. 用户点击一个国家按钮	4. 国家的货币名称出现在lblEquiv中
	5. 用户点击cmdCompute按钮	6. 计算出的等价金额出现在lblEqAmount中
	7. 用户点击另一个不同的国家按钮	8. 新的国家货币名称出现在lblEquiv中
		9. 以前选择的选项按钮复位
		10. 设置当前选择的选项按钮
	11. 用户点击cmdCompute按钮	12. 计算出的等价金额出现在lblEqAmount中
替代序列	行动1和3顺序可以颠倒，相应的响应4和6顺序也可以颠倒；行动7和11可以无限次地重复，相应的响应8、9、10和12也可以随之重复	
交叉引用	R3、R4、R5、R7	
后果	cmdClear得到焦点	

（续）

EEUC 6	修改输入	
参与者	用户	
前提	txtDollar有一个非空金额，或选择了一个国家	
类型	二级功能	
描述	用户重新设置输入，开始一次新的事务处理或修改现有输入	
序列	参与者行动：	系统响应：
	1. 用户通过键盘输入一个美元金额	2. 美元金额出现在txtDollar中
	3. 用户选择一个国家	4. 国家的货币名称出现在lblEquiv中
	5. 用户点击cmdClear按钮	6. txtDollar显示空项
		7. 所选国家选项按钮被复位
	8. 用户通过键盘输入新的一个美元金额	9. 新的美元金额出现在txtDollar中
	10. 用户点击一个不同的国家按钮	11. 国家货币名称出现在lblEquiv中
		12. 设置当前选择的选项按钮
	13. 用户点击cmd Clear按钮	14. txtDollar显示空项
		15. 所选国家选项按钮被复位
替代序列	行动8、10和13可以无限次地重复，相应的响应9、11、12、14和15也可以随之重复	
交叉引用	R3，R4，R6	
后果	txtDollar得到焦点	

EEUC 7	异常情况：没有选择国家	
参与者	用户	
前提	dollarInput得到焦点	
类型	隐藏功能	
描述	用户输入美元金额，并在没有选择国家的情况下点击cmdCompute按钮	
序列	参与者行动：	系统响应：
	1. 用户通过键盘输入一个美元金额	2. 美元金额出现在txtDollar中
	3. 用户点击cmdCompute按钮	4. 出现“必须选择一个国家”消息框
	5. 用户关闭消息框	6. 不再显示消息框
替代序列	-	
交叉引用	R3，R5	
后果	txtDollar得到焦点	

EEUC 8	异常情况：没有输入美元金额	
参与者	用户	
前提	dollarInput得到焦点	
类型	隐藏功能	
描述	用户选择一个国家，并在没有输入美元金额的情况下点击cmdCompute按钮	
序列	参与者行动：	系统响应：
	1. 用户点击一个国家按钮	2. 货币名称出现在lblEquiv中
	3. 用户点击cmdCompute按钮	4. 出现“必须输入一个美元金额”消息框
	5. 用户关闭消息框	6. 不再显示消息框
替代序列	—	
交叉引用	R3，R5	
后果	txtDollar得到焦点	

（续）

EEUC 9	异常情况：既没有输入美元金额也没有选择国家	
参与者	用户	
前提	dollarInput得到焦点	
类型	隐藏功能	
描述	用户在没有输入美元金额和选择国家的情况下点击cmdCompute按钮	
序列	参与者行动：	系统响应：
	1. 用户点击cmdCompute按钮	2. 出现"必须输入一个美元金额并选择一个国家"消息框
	3. 用户关闭消息框	4. 不再显示消息框
替代序列	—	
交叉引用	R5	
后果	txtDollar得到焦点	

20.1.8 真实用例

在Larman的术语中，真实用例与扩展基本用例只有微小的差别。诸如"输入一个美元金额"这样的短语，必须用更具体的"在txtDollar中输入125"替代。类似地，"选择一个国家"要由"点击optBrazil按钮"替代。为了节省篇幅（并不使读者感到厌烦），这里没有给出真实用例。请注意，系统级测试用例可以机械地从真实用例中导出。

20.2 基于UML的系统测试

本书的讨论使我们能够在系统级测试上非常具体，对于GUI应用程序，至少有四个可标识的层次具有相应的覆盖指标。其中两个层次很自然地依赖于UML规格说明，另外两个层次已经在第19章讨论过了。

第一个层次，是测试Larman UML方法第一步中给出的系统功能，这些功能在扩展基本用例中被交叉引用，因此可以很容易地构建类似表20-2那样的关联矩阵

表20-2　用例与系统功能的关联

EEUC	R1	R2	R3	R4	R5	R6	R7
1	X	—	—	—	—	—	—
2		X	—	—	—	—	—
3	—	—	X	X	X	—	—
4	—	—	X	X	X	—	—
5	—	—	X	X	X	—	X
6	—	—	X	X	—	X	X
7	—	—	X	—	X	—	—
8	—	—	X	—	X	—	—
9	—	—	—	—	X	—	

研究一下这个关联矩阵，可以发现有多种可能的方式可以覆盖七个系统功能。一种方式是通过对应于扩展基本用例1、2、5和6的真实用例导出测试用例，需要采用与扩展基本用例相对的真实用例。不同之处是使用具体国家和美元金额，而不是较高层次的描述，例如“点击一个国家按钮”并输入一个美元金额。从真实用例中导出系统测试用例是机械式的：用例前提是测试用例的前提，参与者行动序列和系统响应可直接映射到用户输入事件和系统输出事件序列。扩展基本用例1、2、5和6集合，是回归测试用例的一个很好的例子，因为EEUC5覆盖四个系统功能，EEUC6覆盖三个系统功能。

第二个层次是通过所有真实用例开发测试用例。假设客户对最初的扩展基本用例感到满意，这是最低限度的系统测试覆盖。以下是以扩展基本用例EEUC3为基础的真实用例导出的系统级测试用例示例。

RUC 3	正常使用（首先输入美元金额）	
参与者	用户	
前提	txtDollar得到焦点	
类型	基本功能	
描述	用户输入10美元并选择欧共体，应用程序计算并显示等价金额为9.30欧元	
序列	参与者行动：	系统响应：
	1. 用户通过键盘输入10	2. 10美元出现在txtDollar中
	3. 用户点击欧共体按钮	4. 欧元出现在lblEquiv中
	5. 用户点击cmdCompute按钮	6. 9.30出现在lblEqAmount中
替代序列	行动1和3顺序可以颠倒，相应的响应4和6顺序也可以颠倒	
交叉引用	R3，R4，R5	
后果	cmdClear得到焦点	

SysTC 3	正常使用（首先输入美元金额）	
测试操作者	Paul Jorgensen	
前提	txtDollar得到焦点	
类型	基本功能	
测试操作序列	测试人员输入：	预期系统响应：
	1 通过键盘输入10	2 观察10美元出现在txtDollar中
	3 点击欧共体按钮	4 观察欧元出现在lblEquiv中
	5 点击cmdCompute按钮	6. 观察9.30出现在lblEqAmount中
后果	cmdClear得到焦点	
测试结果	通过/失败	
运行日期	2002年1月23日	

第三层是通过有限状态机导出测试用例，有限状态机由GUI外观有限状态机描述导出，如图20-3所示（与第19章中的框图相同）。通过这种方式得到的测试用例是一个环路（即路径的起始点也是终点，通常是空闲状态）。表20-3给出了九个这类测试用例，其中的编号表示测试用例经过状态的顺序。还有很多其他测试用例，不过这已经能够说明如何找出这些测试用例。

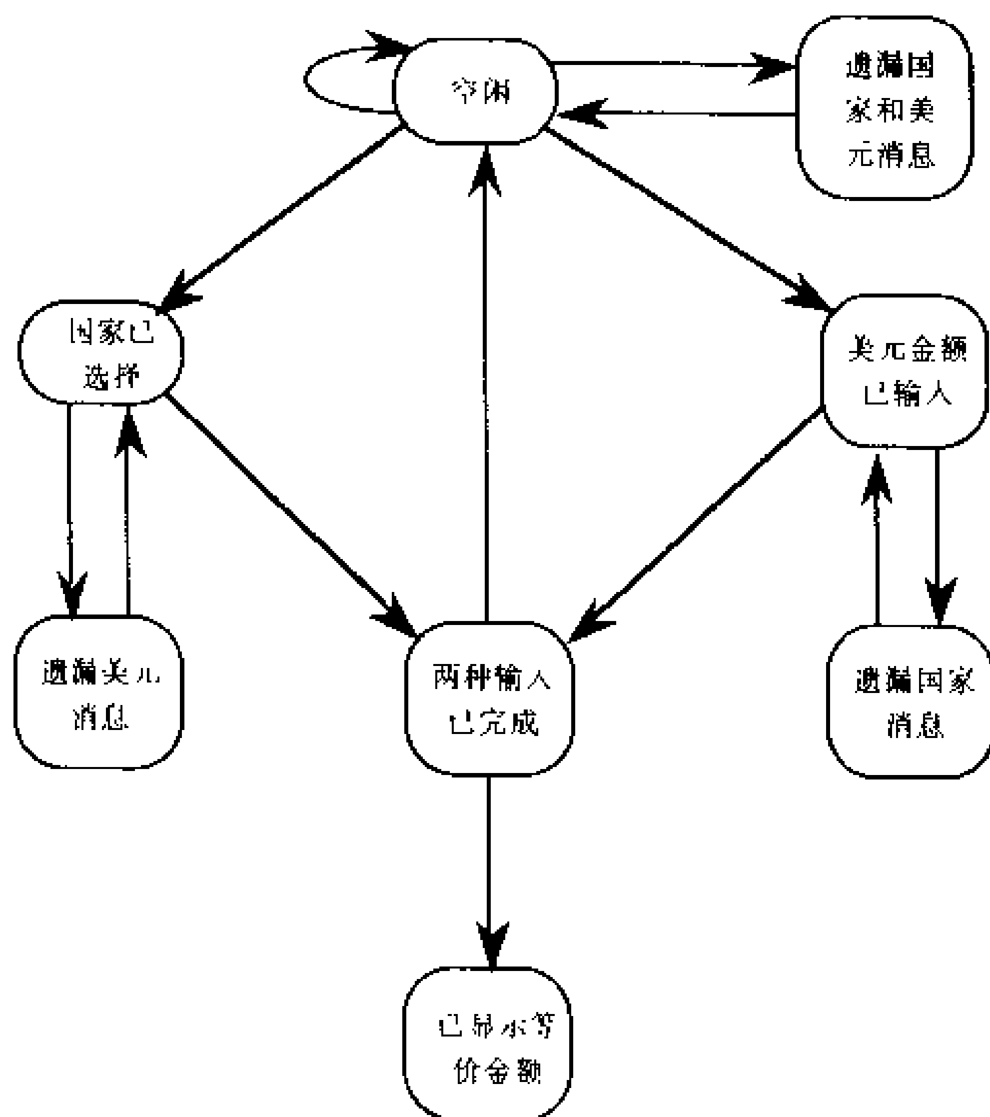


图20-3 GUI有限状态机

表20-3 通过有限状态机导出的测试用例

状态	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9
空闲	1	1	1	1	1	1	1	1	1，3
遗漏国家和美元金额消息								2	2
选择国家		2	2，4			2			4，6
输入美元金额				2	2，4		2		
遗漏美元金额消息			3						5
完成两种输入		3	5	3	5	3	3		7
遗漏国家消息					3				
显示等价金额		4	6	4	6				
空闲	2	5	7	5	7	1	1	1	1

第四层是通过基于状态的事件表导出测试用例（请参阅第19章的表19-1和图19-2）。这种工作必须对每个状态重复进行。我们可以把这一层叫做穷尽层，因为要对每个状态执行所有可能的事件。但是这一层并不真正是穷尽的，因为并没有测试所有跨状态的事件序列。另一个问题是，它是系统测试的极为详细的视图，很可能与集成级测试用例甚至单元级测试用例有很大重复。

20.3 基于"状态图"的系统测试

进行这种测试需要注意。第19章曾经介绍过，"状态图"是系统测试的很好的基础。问题是UML将"状态图"规定为类级。合成多个类的"状态图"得到一个系统级"状态图"并不容易。一种可行的方法，是将每个类级"状态图"转换为一组EDPN，然后像在第15章中介绍的那样，合成EDPN。

20.4 参考文献

Larman, Craig, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice-Hall, Upper Saddle River, NJ, 1998

Zak, Diane, *Programming with Microsoft Visual Basic 6.0-Enhanced Edition*, Course Technology, Inc., 2001.

［Ｇｅｎｅｒａｌ Ｉｎｆｏｒｍａｔｉｏｎ］
书名＝软件测试
作者＝
页数＝１０００
ＳＳ号＝０
出版日期＝

封面页
书名页
版权页
前言页
目录页
第一部分　数学背景
第１章　测试概述
１．１　基本定义
１．２　测试用例
１．３　通过维恩图理解测试
１．４　标识测试用例
１．４．１　功能性测试
１．４．２　结构性测试
１．４．３　功能性测试与结构性测试的比较
１．５　错误与缺陷分类
１．６　测试级别
１．７　参考文献
１．８　练习
第２章　举例
２．１　泛化的伪代码
２．２　三角形问题
２．２．１　问题陈述
２．２．２　讨论
２．２．３　传统实现
２．２．４　结构化实现
２．３　ＮｅｘｔＤａｔｅ函数
２．３．１　问题陈述
２．３．２　讨论
２．３．３　实现
２．４　佣金问题
２．４．１　问题陈述
２．４．２　讨论
２．４．３　实现
２．５　ＳＡＴＭ系统
２．５．１　问题陈述
２．５．２　讨论
２．６　货币转换器
２．７　土星牌挡风玻璃雨刷
２．８　参考文献
２．９　练习
第３章　测试人员的离散数学
３．１　集合论
３．１．１　集合成员关系
３．１．２　集合定义
３．１．３　空集
３．１．４　维恩图
３．１．５　集合操作
３．１．６　集合关系
３．１．７　子集划分
３．１．８　集合恒等式
３．２　函数
３．２．１　定义域与值域
３．２．２　函数类型
３．２．３　函数合成
３．３　关系

３．３．１　集合之间的关系
３．３．２　单个集合上的关系
３．４　合题逻辑
３．４．１　逻辑操作符
３．４．２　逻辑表达式
３．４．３　逻辑等价
３．５　概率论
３．６　参考文献
３．７　练习
第４章　测试人员的图论
４．１　图
４．１．１　节点的度
４．１．２　关联矩阵
４．１．３　相邻矩阵
４．１．４　路径
４．１．５　连接性
４．１．６　压缩图
４．１．７　圈数
４．２　有向图
４．２．１　内度与外度
４．２．２　节点的类型
４．２．３　有向图的相邻矩阵
４．２．４　路径与半路径
４．２．５　可到达性矩阵
４．２．６　ｎ－连接性
４．２．７　强组件
４．３　用于测试的图
４．３．１　程序图
４．３．２　有限状态机
４．３．３　Ｐｅｔｒｉ网
４．３．４　事件驱动的Ｐｅｔｒｉ网
４．３．５　状态图
４．４　参考文献
４．５　练习
第二部分　功能性测试
第５章　边界值测试
５．１　边界值分析
５．１．１　归纳边界值分析
５．１．２　边界值分析的局限性
５．２　健壮性测试
５．３　最坏情况测试
５．４　特殊值测试
５．５　举例
５．５．１　三角形问题的测试用例
５．５．２　ＮｅｘｔＤａｔｅ函数的测试用例
５．５．３　佣金问题的测试用例
５．６　随机测试
５．７　边界值测试的指导方针
５．８　练习
第６章　等价类测试
６．１　等价类
６．１．１　弱一般等价类测试
６．１．２　强一般等价类测试
６．１．３　弱健壮等价类测试
６．１．４　强健壮等价类测试

６．２　三角形问题的等价类测试用例
６．３　ＮｅｘｔＤａｔｅ函数的等价类测试用例
６．４　佣金问题的等价类测试用例
６．４．１　输出值域等价类测试用例
６．４．２　输出值域等价类测试用例
６．５　指导方针和观察
６．６　参考文献
６．７　练习
第７章　基于决策表的测试
７．１　决策表
７．２　三角形问题的测试用例
７．３　ＮｅｘｔＤａｔｅ函数测试用例
７．３．１　第一次尝试
７．３．２　第二次尝试
７．３．３　第三次尝试
７．４　佣金问题的测试用例
７．５　指导方针与观察
７．６　参考文献
７．７　练习
第８章　功能性测试回顾
８．１　测试工作量
８．２　测试效率
８．３　测试的有效性
８．４　指导方针
８．５　案例研究
第三部分　结构性测试
第９章　路径测试
９．１　ＤＤ-路径
９．２　测试覆盖指标
９．２．１　基于指标的测试
９．２．２　测试覆盖分析器
９．３　基路径测试
９．３．１　ＭｃＣａｂｅ的基路径方法
９．３．２　关于ＭｃＣａｂｅ基路径方法的观察
９．３．３　基本复杂度
９．４　指导方针与观察
９．５　参考文献
９．６　练习
第１０章　数据流测试
１０．１　定义/使用测试
１０．１．１　举例
１０．１．２　ｓｔｏｃｋｓ的定义-使用路径
１０．１．３　ｌｏｃｋｓ的定义-使用路径
１０．１．４　ｔｏｔａｌＬｏｃｋｓ的定义-使用路径
１０．１．５　ｓａｌｅｓ的定义-使用路径
１０．１．６　ｃｏｍｍｉｓｓｉｏｎ的定义-使用路径
１０．１．７　定义-使用路径测试覆盖指标
１０．２　基于程序片的测试
１０．２．１　举例
１０．２．２　风格与技术
１０．３　指导方针与观察
１０．４　参考文献
１０．５　练习
第１１章　结构性测试回顾
１１．１　漏洞与冗余

１１．２　用于方法评估的指标
１１．３　重温案例研究
１１．３．１　基于路径的测试
１１．３．２　数据流测试
１１．３．３　片测试
１１．４　参考文献
１１．５　练习
第四部分　集成与系统测试
第１２章　测试层次
１２．１　测试层次的传统观点
１２．２　其他生命周期模型
１２．２．１　瀑布模型的新模型
１２．２．２　基于规格说明的生命周期模型
１２．３　ＡＳＴＭ系统
１２．４　将集成测试与系统测试分开
１２．４．１　结构认识
１２．４．２　行为认识
１２．５　参考文献
第１３章　集成测试
１３．１　深入研究ＳＡＴＭ系统
１３．２　基于分解的集成
１３．２．１　自顶向下集成
１３．２．２　自底向上集成
１３．２．３　三明治集成
１３．２．４　优缺点
１３．３　基于调用图的集成
１３．３．１　成对集成
１３．３．２　相邻集成
１３．３．３　优缺点
１３．４　基于路径的集成
１３．４．１　新概念与扩展概念
１３．４．２　ＳＡＴＭ系统中的MM－路径
１３．４．３　MM－路径复杂度
１３．４．４　优缺点
１３．５　案例研究
１３．５．１　基于分解的集成
１３．５．２　基于调用图的集成
１３．５．３　基于MM－路径的集成
１３．６　参考文献
１３．７　练习
第１４章　系统测试
１４．１　线索
１４．１．１　线索的可能性
１４．１．２　线索定义
１４．２　需求规格说明的基本概念
１４．２．１　数据
１４．２．２　行动
１４．２．３　设备
１４．２．４　事件
１４．２．５　线索
１４．２．６　基本概念之间的关系
１４．２．７　采用基本概念建模
１４．３　寻找线索
１４．４　线索测试的结构策略
１４．４．１　自底向上组织线索

１４．４．２　节点与边覆盖指标
１４．５　线索测试的功能策略
１４．５．１　基于事件的线索测试
１４．５．２　基于端口的线索测试
１４．５．３　基于数据的线索测试
１４．６　ＳＡＴＭ测试线索
１４．７　系统测试指导方针
１４．７．１　伪结构系统测试
１４．７．２　运行剖面
１４．７．３　累进测试与回归测试
１４．８　参考文献
１４．９　练习
第１５章　交互测试
１５．１　交互的语境
１５．２　交互的分类
１５．２．１　单处理器中的静态交互
１５．２．２　多处理器中的静态交互
１５．２．３　单处理器中的动态交互
１５．２．４　多处理器中的动态交互
１５．３　交互、合成与确定性
１５．４　客户－服务器测试
１５．５　参考文献
１５．６　练习
第五部分　面向对象的测试
第１６章　面向对象的测试问题
１６．１　面向对象测试的单元
１６．２　合成与封装的涵义
１６．３　继承的涵义
１６．４　多态性的涵义
１６．５　面向对象测试的层次
１６．６　ＧＵＩ测试
１６．７　面向对象软件的数据流测试
１６．８　第五部分采用的例子
１６．８．１　面向对象的日历
１６．８．２　货币转换应用程序
１６．９　参考文献
１６．１０　练习
第１７章　类测试
１７．１　以方法为单元
１７．１．１　ｏ－ｏＣａｌｅｎｄａｒ的伪代码
１７．１．２　Ｄａｔｅ．ｉｎｃｒｅｍｅｎｔ的单元测试
１７．２　以类为单元
１７．２．１　ｗｉｎｄｓｈｉｅｌｄＷｉｐｅｒ类的伪代码
１７．２．２　ｗｉｎｄｓｈｉｅｌｄＷｉｐｅｒ类的单元测试
第１８章　面向对象的集成测试
１８．１　集成测试的ＵＭＬ支持
１８．２　面向对象软件的MM－路径
１８．３　面向对象数据流集成测试框架
１８．３．１　事件驱动和消息驱动的Ｐｅｔｒｉ网
１８．３．２　由继承导出的数据流
１８．３．３　由消息导出的数据流
１８．３．４　分片
１８．４　练习
１８．５　参考文献
第１９章　ＧＵＩ测试

１９．１　货币转换程序
１９．２　货币转换程序的单元测试
１９．３　货币转换程序的集成测试
１９．４　货币转换程序的系统测试
１９．５　练习
第２０章　面向对象的系统测试
２０．１　货币转换器的ＵＭＬ描述
２０．１．１　问题陈述
２０．１．２　系统功能
２０．１．３　表示层
２０．１．４　高层用例
２０．１．５　基本用例
２０．１．６　详细ＧＵＩ定义
２０．１．７　扩展基本用例
２０．１．８　真实用例
２０．２　基于ＵＭＬ的系统测试
２０．３　基于“状态图”的系统测试
２０．４　参考文献
附录页